

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

HADOOP

BIG DATA PROCESSING SYSTEM

大数据处理系统

Hadoop 源代码情景分析

毛德操 ©著



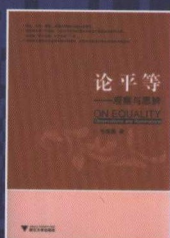
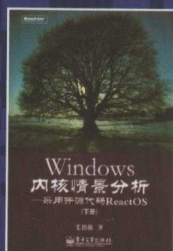
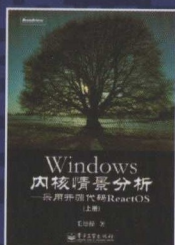
ZHEJIANG UNIVERSITY PRESS
浙江大学出版社

毛德操

浙大网新科技◎首席科学家

连连支付◎大数据与区块链特别顾问

已出版著作



Hadoop 源代码情景分析

 ZHEJIANG UNIVERSITY PRESS
浙江大学出版社

图书在版编目 (CIP) 数据

大数据处理系统: Hadoop 源代码情景分析 / 毛德操

著. —杭州: 浙江大学出版社, 2017. 3

ISBN 978-7-308-16669-0

I. ①大… II. ①毛… III. ①数据处理软件

IV. ①TP274

中国版本图书馆 CIP 数据核字 (2017) 第 022599 号

大数据处理系统: Hadoop 源代码情景分析

毛德操 著

责任编辑 樊晓燕 吴昌雷

责任校对 金佩雯 刘 郡

封面设计 春天书装

出版发行 浙江大学出版社

(杭州市天目山路 148 号 邮政编码 310007)

(网址: <http://www.zjupress.com>)

排 版 杭州中大图文设计有限公司

印 刷 杭州杭新印务有限公司

开 本 787mm×1092mm 1/16

印 张 49

字 数 1223 千

版 次 2017 年 3 月第 1 版 2017 年 3 月第 1 次印刷

书 号 ISBN 978-7-308-16669-0

定 价 128.00 元

版权所有 翻印必究 印装差错 负责调换

浙江大学出版社发行中心联系方式: 0571-88925591; <http://zjdxcs.tmall.com>

前 言

这本书不是为所有想要对大数据有所了解的人而写。但是如果你有点野心,想对大数据处理系统有比较深入透彻的了解,特别是想有朝一日自己也设计一个这样的系统,甚至自己把它写出来,那么你真应该认真读一下这本书,以及 Hadoop 的源代码,看看人家是怎么设计怎么实现的。然后,在最后一章,再看看 Spark 又是什么样的,有些什么改进。你将看到,在一个计算机集群上构筑一个大数据处理系统,哪些成分是必不可少的,哪些方面又是可以改进的,它与操作系统的关系怎样,而作为大规模计算机集群的“操作系统”又可以和应该是什么样的。

不过我也尽力把它写成让非计算机专业的读者也能读懂,当然他们的困难会多一些,但也绝非无法理解。正因如此,我的叙述也许显得过于通俗直白,有时候可能还有点啰嗦。

我之起意写这本书,还是四年以前,当时的 Hadoop 还是 1.x 版,Spark 还没有出来。当时我为连连支付的一群工程师开了个大数据与 Hadoop 的讲座,为要讲对并讲清,就得去抠源代码,既然这样,那何不干脆写成一本书?然而,此后 Hadoop 的源码发生了堪称翻天覆地的变化,从 1.x 版到 2.0 版是个重起炉灶式的改写,这让我对 1.x 版所做的研究几乎全都付诸东流。无奈之下,我也只得跟着重起炉灶。所以这本书的写作经历了那么长的时间,幸而现在终于要付印了。

除我本人的努力外,这本书的出版首先要感谢浙江大学出版社的樊晓燕和吴昌雷两位编辑,没有两位编辑的热情支持和辛勤劳动,本书的出版显然还要困难得多,他们那一丝不苟的风格也使我深受教益。

尤其要感谢的是我的恩师何志均教授。记得那时候我去看他,当时他身体尚好,聊起了刚刚兴起的大数据热潮,自然也就聊到了 Hadoop。何老师说他的一个学生(杨小虎教授)对他说要像当年读 Unix 源码那样组织学生读 Hadoop 源码,我就说了想要写作本书的念头。他听后对我大加鼓励说,对对,你快把它写出来,就像那本 Linux 内核一样。后来,我去看他的时候他又曾问起写得怎么样了,那次还聊到了 Spark,以及 Spark 与 Hadoop 间的竞争和比较。种种情景宛在眼前,但是何老师已经仙逝。我这一生中,最于我有恩的人就是何老师了。不是何老师的爱护和提携,我一个插队回城的超龄(已过当时报考大学的年龄限制)知青根本不会进计算机这个门,也进不了这个门,那也就不会有现在这本书了。

另外,我的几位老朋友也在本书写作的过程中给了我许多鼓励。胡希明教授就不用说了,

我们两人同在连连支付担任顾问,常在一起喝茶闲聊或作不闲之聊,中间他给过我多少鼓励和启发也记不清说不明了。还有另一位王泽兵教授,当时在浙大城市学院当系主任,也曾邀请我去做过一次讲座,那也促使我萌生了写作本书的意向。当然还有别的朋友,特别是“弯曲的数据”微信群中的朋友们,这里就不一一细数了。这里,我要对胡老师、王老师等朋友一起致以真诚的谢意。

最后,关于本书内容的正确性和完整性,我自知水平有限,写了这么厚的一本书,要说没有错误我自己也不信,但是我又确实不知道错误在哪,凡是知道的都已改过来了,剩下的就是自己茫然无知的错误。现在这本书的内容是基于 Hadoop 2.6 版的,我已努力把这个版本的方方面面都写进本书,但实际上当然不可能面面俱到。所以,我衷心欢迎读者发现和指正我的错误,补充我的疏漏。

毛德操

2017 年 3 月于杭州

目 录

第 1 章 大数据与 Hadoop	1
1.1 什么是大数据	1
1.2 大数据的用途	3
1.3 并行计算	7
1.4 数据流	8
1.5 函数式程序设计与 Lambda 演算	12
1.6 MapReduce	15
1.7 大数据处理平台	17
1.8 Hadoop 的由来和发展	17
1.9 Hadoop 的 MapReduce 计算框架	19
1.10 Hadoop 的分布式容错文件系统 HDFS	20
第 2 章 研究方法	22
2.1 摘要卡片	22
2.2 情景分析	27
2.3 面向对象的程序设计	27
2.4 怎样阅读分析 Hadoop 的代码	30
第 3 章 Hadoop 集群和 YARN	31
3.1 Hadoop 集群	31
3.2 Hadoop 系统的结构	40
3.3 Hadoop 的 YARN 框架	42
3.4 状态机	45
3.5 资源管理器 ResourceManager	68
3.6 资源调度器 ResourceScheduler	72
第 4 章 Hadoop 的 RPC 机制	74
4.1 RPC 与 RMI	74

4.2	ProtoBuf	96
4.3	Java 的 Reflection 机制	104
4.4	RM 节点上的 RPC 服务	105
4.5	RPC 客户端的创建	111
第 5 章	Hadoop 作业的提交	127
5.1	从“地方”到“中央”	127
5.2	示例一:采用老 API 的 ValueAggregatorJob	128
5.3	示例二:采用新 API 的 WordCount	138
5.4	示例三:采用 ToolRunner 的 QuasiMonteCarlo	142
5.5	从 Job. submit() 开始的第二段流程	148
5.6	YARNRunner 和 ResourceMgrDelegate	165
第 6 章	作业的调度与指派	182
6.1	作业的受理	182
6.2	NM 节点的心跳和容器周转	199
6.3	容器的分配	205
第 7 章	NodeManager 与任务投运	219
7.1	AMLauncher 与任务投运	219
7.2	MRAppMaster 或 AM 的创建	223
7.3	资源本地化	234
7.4	容器的投运	249
第 8 章	MRAppMaster 与作业投运	261
8.1	MRAppMaster	261
8.2	App 资源与容器	271
8.3	容器的跨节点投送和启动	283
8.4	目标节点上的容器投运	286
8.5	Uber 模式下的本地容器分配与投运	293
8.6	任务的启动	295
8.7	MapTask 的运行	301
8.8	ReduceTask 的投运	303
第 9 章	YARN 子系统的计算框架	307
9.1	MapReduce 框架	307
9.2	Streaming 框架	317

9.3 Chain 框架	329
9.4 Client 与 ApplicationMaster	335
第 10 章 MapReduce 框架中的数据流	348
10.1 数据流和工作流	348
10.2 Mapper 的输入	350
10.3 Mapper 的输出缓冲区 MapOutputBuffer	360
10.4 作为 Collector 的 MapOutputBuffer	365
10.5 环形缓冲区 kvbuffer	369
10.6 对 MapOutputBuffer 的输出	373
10.7 Sort 和 Spill	376
10.8 Map 计算的终结与 Spill 文件的合并	380
10.9 Reduce 阶段	389
10.10 Merge	399
10.11 Reduce 阶段的输入和输出	407
第 11 章 Hadoop 的文件系统 HDFS	415
11.1 文件的分布与容错	415
11.2 目录节点 NameNode	419
11.3 FSNamesystem	423
11.4 文件系统目录 FSDirectory	426
11.5 文件系统映像 FsImage	433
11.6 文件系统更改记录 FSEditLog	446
11.7 FSEditLog 与 Journal	457
11.8 EditLog 记录的重演	460
11.9 版本升级与故障恢复	464
第 12 章 HDFS 的 DataNode	477
12.1 DataNode	477
12.2 数据块的存储	481
12.3 RamDisk 复份的持久化存储	493
12.4 目录扫描线程 DirectoryScanner	501
12.5 数据块扫描线程 DataBlockScanner	511
第 13 章 DataNode 与 NameNode 的互动	519
13.1 DataNode 与 NameNode 的互动	519
13.2 心跳 HeartBeat	526

13.3	BlockReport	539
第 14 章	DataNode 间的互动	559
14.1	数据块的接收和存储	559
14.2	命令 DNA_TRANSFER 的执行	588
第 15 章	HDFS 的文件访问	592
15.1	DistributedFileSystem 和 DFSClient	592
15.2	FsShell	594
15.3	HDFS 的打开文件流程	599
15.4	HDFS 的读文件流程	604
15.5	HDFS 的创建文件流程	610
15.6	文件租约	621
15.7	HDFS 的写文件流程	624
15.8	实例	639
第 16 章	Hadoop 的容错机制	642
16.1	容错与高可用	642
16.2	HDFS 的 HA 机制	648
16.3	NameNode 的倒换	664
16.4	Zookeeper 与自动倒换	670
16.5	YARN 的 HA 机制	677
第 17 章	Hadoop 的安全机制	680
17.1	大数据集群的安全问题	680
17.2	UGI、Token 和 ACL	690
17.3	UGI 的来源和流转	698
17.4	Token 的使用	703
第 18 章	Hadoop 的人机界面	709
18.1	Hadoop 的命令行界面	709
18.2	Hadoop 的 Web 界面	714
18.3	Dependency Inject 和 Annotation	727
18.4	对网页的访问	730
第 19 章	Hadoop 的部署和启动	741
19.1	Hadoop 的运维脚本	741

19.2 Hadoop 的部署与启动	743
19.3 Hadoop 的日常使用	749
19.4 Hadoop 平台的关闭	752
第 20 章 Spark 的优化与改进	754
20.1 Spark 与 Hadoop	754
20.2 RDD 与 Stage——概念与思路	754
20.3 RDD 的存储和引用	757
20.4 DStream	758
20.5 拓扑的灵活性和多样性	759
20.6 性能的提升	762
20.7 使用的方便性	763
20.8 几个重要的类及其作用	766
参考资料	773

第 1 章

大数据与 Hadoop

1.1 什么是大数据

所谓“大数据(big data)”,首先当然是数据。“大”是形容词,是对于“数据”这个词的限定和修饰。而我们所讲的“数据(data)”,则是指对于信息的数字化记录。

客观世界无时不在产生信息,每一种物理的、化学的、生物的、思想的、行为的现象和过程都带有大量的信息。如果不加记录,这些信息就都流逝了。其中也有极少数的信息会被大自然以某种方式保存下来,例如化石,但是从数量比例上看那是可以忽略不计的。也有些信息会被人们所记忆并在一段时间内口耳相传,可是如果不加以进一步记录,也就慢慢湮没了。

文字的发明之所以重要,就是因为它提供了一种记录信息的手段。当然,单纯的记录并不解决问题,记录下来的信息还应可以回放、可以复制转录、可以长期保存,还要可以检索、可以比对。文字在一定程度上满足了这些要求:人们通过阅读在头脑中“回放”和恢复被记录成文字的信息;通过抄写和印刷加以复制;通过纸张、竹简、羊皮、石壁等物理介质解决长期保存的问题;通过顺序扫描、分册、分段、加签、索引等手段和方法解决检索的问题。

但是,从总体上看,人工的文字记载是一种效率很低的记录方法。正因为如此,人类有史以来所积聚的书籍数量也很有限。据称美国国会图书馆的藏书量为 3000 万册(尽管其书架总长达 800 公里),若按平均每册信息量 1MB 计算(按文字信息,不是按扫描影像),总共也才 30TB 的信息量。哪怕再乘上 10 倍,也才 300TB,即 0.3PB。这样的量,放在现在虽然也能算得上“大数据”,但真的也不算什么。相比之下,如今纽约股票交易所每天产生的交易数据据说有 1TB,一年下来的信息量就超过美国国会图书馆的馆藏了。至于大学的图书馆,哈佛大学的藏书量是最大的,也只有 1500 多万册(且不说这里面还有重复),那就是说充其量也不会超过 150TB 的信息量。

而且文字记载还有失真的问题。因为文字是人在其所见所闻的基础上经过头脑加工处理以后才写出来的,人的观察本已难免片面和失真,头脑的加工更有扭曲的可能,表达能力也受到种种限制,所以文字记载往往流于抽象、片面和失真。

照相术、录音技术、录像技术的出现使人类记录信息的效率和保真度都得以大幅提高。保真度高了,所记录内容的权威性也就自然提高了,所以一段两分钟的录像带很可能会压倒几十个人的书面证词。

然而早期的文字、照相、录音、录像,即使电子化之后,也都是基于模拟技术的,这样的技术有很多缺点。首先是不便甚至不能用机器进行检索和比对,因为模拟技术所记录的是波形,

这些波形必须得回放出来为人所听闻和目睹,经过大脑的加工处理才能获取其真正的内容,那就是大致等价于假定观察者当初正好处于摄像机(传感器)所在位置和角度的所见所闻。另一方面,模拟信号的复制也是个问题,因为每次复制都会引入一定程度的失真。这种失真也影响到记录的长期保存。由于物理介质都是有一定寿命的,过一段时间就得把内容翻录到另一个具体的介质上,如果每次翻录都要引入失真,那么几次翻录以后的信号质量就不行了。

所以数字技术的出现有着划时代的意义。当然,数字技术的出现和发展与计算机技术的发展有密不可分的关系。有了数字化的电子记录技术以及将模拟信号数字化的技术之后,首先在复制过程中就不再有失真的问题了,这一点本身就很重要。更重要的是,数字化的记录便于机器处理,包括检索和比对,这一点对于文字记录尤为重要。目前对于文字信息的数字化主要有两种方法:一种是重新输入,就是照着文本在键盘上再输入一遍;另一种是采用 OCR (optical character recognition, 光学字符识别)技术,扫描文字图像并加以识别。把文字信息数字化以后,用机器进行搜索比对就很方便了。不过,要从数字化了的一般图像和音频信号中识别非文字的内容,则仍很困难,那属于模式识别,也是现在很活跃的一个研究方向。尽管如此,有了数字化记录技术,情况毕竟还是大不一样了。就说对于图像和音频的内容识别技术的研究,要是没有数字化技术那是根本就无法开展的。

所以,前面那句话,“我们所讲的数据是指对于信息的数字化记录”,就容易理解了。光是把信息记录下来还不一定就能成为数据,必须是数字化的记录才算。

“数据”一词在英文里是 data,这是个复数名词,其单数形式是 datum,但是一般都只用其复数形式 data,除非要特别强调所说的乃是单项数据。所以,人们在使用 data 这个词的时候,常常是把它看成一个集合的。正是在这个意义上,才有了“big data”即“大数据”的说法。因为 data 是复数,这就不是指单项数据的数值之大,而是指“data”这个集合之大,即集合中元素的数量之大。

由此可见,所谓大数据,首先就是指数量之大。但光是数量大还不足以概括“大数据”这个词的意义。有人把大数据的特征总结成三个以 V 字母开头的英文单词,即 volume(数量)、variety(多样性)和 velocity(产生的速度)。

首先是数量,这已无须多说,一般认为达到 TB 甚至 PB 的规模才能称得上是“大数据”(1PB=1024TB, 1TB=1024GB, 1GB=1024MB。例如:我的笔记本电脑,其硬盘容量只是 300GB 左右,即 0.3TB)。不过,业界也并没有规定一个很具体而明确的判断标准,这是个比较模糊的概念。

然后是多样性,这很重要。所谓数据的多样性,是指来源和内容的多样性,以及结构和格式的多样性。首先是来源,人们常说“兼听则明,偏信则暗”,单一来源的数据未必真实可靠。还有就是空间跨度和时间跨度的问题。先说空间跨度,首先是地域(和人群):你的数据只是一个省的,他的数据虽然数量上不如你,却覆盖了 10 个省,你说谁的数据更有价值? 空间跨度也不仅仅是地域的问题,同时也是领域的问题:你手里只有商品交易数据,他的数据虽然数量上不如你,却有商品交易、医疗、教育、户籍等多个方面的数据,里面或许还有一些水文资料(虽说貌似没有多大用处),你说谁的数据更有价值? 再说时间跨度:你只有一年的数据,他的数据虽然数量上少一些,却覆盖了前后 10 年,你说谁的数据更有价值? 所以多样性的问题实际上关系到数据的质量,不能脱离多样性而孤立地强调数量。另一方面,数据的多样性也带来大数据处理技术上的许多特点:因为来源不一,就不可能全都符合某种格式和结构,更多的数据可能

是无格式的“非结构化”数据;因为空间跨度大,可能涉及许多不同领域,往往就无法事先设计好一种普遍适用的数据库模式。即使属于同一种性质的数据,处理时的计算方法和流程也可能会有不同,比方说在有些数据里热量以“大卡”为单位,有些却以“千焦”为单位。如此等等,不一而足。

还有数据产生的速度。如前所述,如果按文字(而不是按扫描后的影像)计算,美国国会图书馆的信息藏量满打满算也不过 30 ~ 300TB。当然不能说人类有史以来的全部文字记录皆备于此,但是这个图书馆的收藏确实已经占了相当大的比例。如果按两千年的时间跨度,那么平均每年产生的文字记录不超过 150GB。然而,现今光是纽约股票交易所每天产生的交易数据就可达到 1TB。这二者的产生速度,显然不可同日而语。所谓“快”,就是单位时间内的数据量可以很大。数量的大小是个相对的概念,如果现有的系统对于每天产生的数据都能轻松应付,那么哪怕绝对量再大,也不能算大。而如果数据有时候会像潮水一样涌来,换用再大的系统也觉难以应付,那就真的是大了;要是数据每天都像潮水一样涌来,那就更大了。所以,当我们说“大数据”时,也包含了产生速度有时候会很快这么一层意思。数据来得快了,对于处理、分析的实时性甚至可行性都带来了挑战。

有人在这 3V 特征的基础上又加了一个 V,即 value,意思是单项数据的价值往往不大,但是作为一个集合则价值巨大。不过这个 V 似乎有点牵强,价值究竟大不大要看挖掘的结果,在此之前我们只是相信它潜在的价值可能很大。

还有人又加上了另一个 V,即 veracity,意思是说数据的真实性,因为这些数据大多都是第一手的原始数据。但是这同样也很牵强,事实上也有人加上了另一个 V,即 variability,意思是说数据质量的变动很大,需要去伪存真、沙里淘金。

综上所述,我们不妨用“多、杂、快、乱”四个汉字概括大数据的特征,其中“多”是主要特征。如果数量不大,哪怕再杂、再乱、来得再快,也称不上大数据。

那么一般所谓的“大数据”都有些什么来源呢?最主要的当然是业务数据,比方说商品交易数据主要来自商店的台账或电子商务的交易记录;再如日内瓦的大型强子对撞机,据说每年能产生高达 15PB 的数据。其次是业务活动中产生的日志(log)信息和统计信息,这主要产生于一些运营商、大型网站等。另一个重要的来源就是纸质材料经过 OCR 识别或重新输入而得的数字化文字信息,如文件、合同、协议、备忘录、信件、报刊、书籍、文告、法律等。当然,互联网也是个重要的数据来源,无论是通过网页搜索爬取,还是网上下载,都能得到不少数据。不言而喻,正在方兴未艾的物联网,即传感器网络,将又是一个重要的数据来源。可以设想一下,像杭州这样一个城市,每天从各个地点和角落的摄像头能产生多少数据?还有外购,也是一种数据来源。再如社会调查,也会产生不少数据。

显然,想要列出一个完整的数据来源清单是不现实的,但是主要的也就是上述这些来源。

1.2 大数据的用途

有了大数据之后,我们能对这些数据做些什么,能拿它来做什么用,又为什么重要呢?概言之,大数据可以帮助我们更好地了解和认识客观世界,从而使我们可以更好地适应、利用和改造客观世界。

我们对客观世界的认识来自对客观存在的反映,是对来自客观世界的各种刺激的反应。

客观存在不仅是自然的、物质的,也是社会的、精神的,别人的主观感受(如时尚)对我们来说也是一种客观存在。而数据,就是对客观存在的记录和反映。除亲身感受之外最好的办法就是通过数据了解世界。但是单项的数据只能是片面的、表面的、个别的,这就需要我们z把数据关联和整合起来。

在某种意义上,人类对于客观世界的认知是一个自觉或不自觉、精确或模糊、彻底或不彻底地运用贝叶斯方法求解逆向概率的过程。贝叶斯是 18 世纪的一个英国业余数学家。所谓贝叶斯方法,一言以概之就是从某种认知(人的认知一定带有主观性)开始,不断根据新的观察所得更新我们的认知,以逼近客观真实并减小不确定性。显然,这是一种反映论的认知过程。后来法国数学家拉普拉斯将其叙述为求逆向概率的过程,并提出了“贝叶斯公式”(其实应该称为“贝叶斯—拉普拉斯公式”)。什么叫逆向概率呢?我们举个例子:假定一个袋子里有 100 个小球,其中 70 个是白的、30 个是黑的,现在随机从袋子里摸取,问摸得白球的概率是多少?这是正向的概率问题。现在把问题变一下。我们不知道袋子里那些球的颜色,但是摸了 5 次是 3 白 2 黑,于是我们就猜只有这么两种颜色,而且白和黑的比例是 3 : 2,可是这个猜测正确的概率有多大呢?然后再摸两次都是白的,那么原先猜测的比例即 3 : 2(仍有可能)正确的概率是多少?而 5 : 2 的概率又是多少?这就是逆向概率的问题。联想一下市场调查、精准营销,我们不就是在根据销售数据来推断顾客对某些商品或服务的喜好程度吗?不仅如此,我们在科学研究中采用的归纳法,其实也是这么个过程。当然,这些过程中也要用到正向概率,贝叶斯公式本身就是建立在正向概率和条件概率基础上的。

但是我们不可能把所有的数据都拿来用于某种特定逆向概率的计算,而只能采用那些相关的数据;不相关的数据就只能暂时留作别用,也许某一天会用于别的逆向概率的计算,或发现其实存在着某种相关。而且,我们也不一定是先有了计算某个概率的明确目标而去筛选相关的数据;有时候倒是反过来,先发现某些数据之间有关联,才想到这里存在着某种可能性、某种逆向概率。所以首先我们得知道哪些数据是互相关联的。所谓数据间的相关性(correlation),就是数据之间有什么内在联系(linkage)、有什么关系(relationship)的意思。从数据中寻找、发现、验证、利用(尤其是用于逆向概率的计算)相关性,就称为“数据挖掘”。实际上,逆向概率本身就代表着一种相关性,而不是因果性。人们在决策的时候当然最好是根据因果性,就像我们运用物理定律一样,然而有很多现象我们一时还不知道因果,可是却知道某些现象之间相关,在这样的情况下就只能退而求其次地根据相关性决策,而且这往往也确实有效。比方说有许多民间验方,我们并不知道其所以能治病的原因,但是很多人试过了,证明这个方子与某种疾病之间确实相关,按这个方子服药病就好了,那么我们就根据这个相关性决策服药。尽管此时我们并不知道这里面的原因,但却知道情况确实如此。

Jeffrey Ullman 教授(他在编译原理、计算理论、数据库方面的著作都已成为经典)在 *Mining of Massive Datasets* 一书中说,对于“数据挖掘”最广泛接受的定义是:在数据中发现模型(the discovery of “models” for data)。我们不妨把这句话改成更容易理解的“在数据中发现规律”。但是一下子就要“发现规律”似乎要求太高了,“发现相关性”应该更实际一些。

既然我们人的认知及其与数据的关系是这样的,那么实现“人工智能”的机器也大体如此,因为现在的人工智能都是对人的模仿。而“机器学习”,则是对人的认知过程的模仿。拿大量数据样本来“训练”机器,就是在模仿人从小接收大量外界刺激以获得知识的过程。可以推测,机器学习的过程一定隐含着某种形式的数据挖掘。试想我们拿 1 亿个样本来训练一个神经元

网络,如果这1亿个样本都是随机生成的,互相之间毫不相关,就像“白噪声”一样,那这个机器能学到什么?机器之所以可训练,根本的原因就是用于训练的样本之间存在相关性。其实人也一样,如果我们在训练(教育)一个小孩时,今天这样说,明天又那样说,前言不搭后语,也没有个一以贯之的逻辑,那么这个孩子非被训练傻了不可。

由此可见,从数据里面寻找、发现、验证相关性,并加以利用,是人们处理和利用大数据的核心所在。

人们对于大数据的处理大体上可以分成四个方面:业务处理、常规统计分析、数据挖掘和机器学习。

1. 业务处理

业务处理一般是实时的 OLTP,即“在线事务处理(online transaction processing)”。比方说,一个计算机集群与证券交易所连线,实时对发生的证券交易记录进行分析,立即自动做出买进或卖出的决策和操作,这样的处理就是 OLTP。

2. 常规统计分析

常规统计分析一般是指那些事先(数据产生和采集之前)就有计划要进行的、设计意图之内的统计分析。正因为是事先计划好的,所以采集的数据通常是有结构、有格式的,相应的数据库也有事先设计好的“模式(schema)”,并且一般都采用关系式数据库。至于具体的统计项目,则大多是些常规的内容,包括,比方说,中值、均值、分布、标准差等。对于这些事先计划好的常规统计分析,(相同性质、相同结构的)数据量增大的意义在于样本数的增大,而样本数增大则更能满足“大数定理”的条件,使统计结果更加精确和可信。但是,单纯的常规统计分析一般不大会使人有意外的发现。

3. 数据挖掘

数据挖掘(data mining)也是一种统计分析,或者依赖于统计分析。但是,与常规统计分析不同,数据挖掘的目标常常是“计划外”的,而挖掘的目的是想要在数据中寻找和发现(原先不明确知道的)相关性,所以最感兴趣的统计项目往往就是相关系数。当然,具体的计算可能也会包括均值、分布、标准差等,但那都是为发现相关性和逆向概率服务的。不过也有些时候相关性本身是明白无误的,而目的则变成了依此将不同来源的数据整合在一起,相当于关系数据库中的 join 操作,用碎片拼凑全貌。可以说,想要从大数据获益,在很大程度上要靠意外发现,而意外发现主要来自数据挖掘。数据挖掘可以是在线的,但一般都不是实时的,往往跟具体业务的进行没有直接的关系,目的只在于对已有数据的分析,所以称为 OLAP,即“在线分析处理(online analytical processing)”。其实,数据挖掘也常常是离线(offline)的,或者说是关起门来进行的。不过“数据挖掘”这个词的使用也并非很严格,例如人们常常把从大量数据中过滤出所需数据也看成挖掘。

4. 机器学习

如前所述,机器学习(machine learning)是一种人工智能方法,目的是通过使用大量数据的“训练”使机器(计算机)形成智能,用于计算机辅助或自动判断决策。人工智能的形成有两种方法,或者说两种流派。一种是对人类大脑的模仿,这种方法着重于推理,在学习的过程中从训练数据中抽取特征以形成基础事实和推理规则。有了推理规则之后,遇到有数据输入就运用这些规则进行推理,以得出判断和决策,同时也使推理规则进一步完善和准确。具体采

用的是知识库、规则库、推理引擎之类的传统技术。另一种是对人类小脑的模仿,这种方法着重于使机器通过训练形成类似于“条件反射”和直觉那样的机制,而并不依赖于显式的推理。这就好像人类产生条件反射时对外界刺激的反应是“下意识”的,并经过大脑思考推理,所以反应速度很快。具体采用的技术则是神经网络,特别是“深度学习”的深层神经网络。其实,对神经网络的训练相当程度上就是个隐性的发现相关性的过程。

本书主要针对大数据处理平台 Hadoop,虽然在这个平台上也能做一些机器学习的处理,但 Hadoop 主要并不是用来做机器学习的,也不是用于实时业务处理 OLTP 的,而是一个用于 OLAP 的平台。所以我们更关注的是数据挖掘。不过常规统计分析、数据挖掘、机器学习这三者之间并无明确的边界划分,许多计算既可以归入数据挖掘,也可以归入机器学习。其实,从本质上说,Hadoop 就是一个批处理式的并行计算平台,只要是这样的计算都可以在 Hadoop 上进行。所以在 Hadoop 的代码中就有采用蒙特卡洛方法计算圆周率的示例。

如前所述,数据挖掘的核心是发现数据之间,从而客观事物之间的相关性,而且目标常常是不确定的,带有尝试性质的。这一点可以说再怎么强调也不为过。

数据是对于客观世界的反映,是对客观事物的描述和记录。可是没有一种描述和记录能够做到完全和准确,任何一种描述和记录都难免是片面的和失真的。许多本来是互相关联、有很高相关性的信息,到了不同角度、不同视野、不同粒度、不同主观感受的记录中,可能就变成孤立而片面,甚至是扭曲的了。而实际存在的相关性,则就隐藏在这些数据之间。我们在做研究的时候常常要拿不同来源的数据加以参照和引证,就是这个道理。所以,数据间的相关性来源于客观事物本身之间的相关性,而数据挖掘则是企图在尽可能大量、尽可能广泛和多元的数据中发现客观事物之间的联系(如果存在的话),以便尽可能真实地还原客观事物的本来面目,帮助决策和判断。

相比之下,在单一来源的、预先设计好的数据中,就难有互相参照引证的作用,不太容易发现预期之外的相关性了。当然这也不是绝对的,在单一来源的数据中有意外的发现也是有可能的。我们不妨假定有这样一种情景:数据库中有 1000 万人的登记表,其中每个人都有出生年份,还有好多方面的信息,其中有个人的业余爱好。这样的登记表当然是预先设计好的,然而经过一些分析和整理之后我们也许发现这些人的业余爱好随出生年份呈现明显的周期性,比方说哪几年出生的人中喜欢数理的比例偏高,而且以多少年的周期起伏变化,而哪几年出生的人中则喜欢艺术的比例偏高,而且以另一种周期起伏变化。于是我们可能就发现了一种原先不知道的相关性,而且早先设计这种登记表的人也绝没有想到居然会有这样的相关性。至于为什么有这样的相关性,那是另一个有待进一步研究的问题了。同一个数据库中的数据,当然有相关性,但是这里我们却发现了原来不知道的、隐藏着的相关性,所以这也属于数据挖掘。但尽管如此,更多的机会还是发生在多元的数据之间。

“挖掘”这个词,在英文中是 mining,这个词翻译得很传神。本来,mining 就是采矿的意思,矿产不是人类有预谋地埋在那里的,人们只是根据各种迹象猜想或怀疑某处的地下有矿,就来钻探一下试试,发现有戏才大规模加以开采。“挖掘”并不是拿着祖先留下的“藏宝图”去挖宝,而是带有试试看的意味,难免会是东挖挖、西挖挖,这样挖过来、那样挖过去。所以,数据挖掘的过程常常是对同一批数据反复进行不同处理的过程。

作者有个同学从维基百科下载了整个数据库,从中挖掘有记载的上百万人的专业/职业、成就、性格、地位与其星座即出生月日的相关性,并不无幽默地称之为“计算星座学”。初听之

下这似乎“无厘头”，但是其实未必，这里面也可能真有点关系，结论只能产生于调查研究之后。按常理说，人的这些性状与出生月日应该是无关的，因而对于出生月日应该是均匀分布的，但是如果挖掘统计的结果表明确实不均匀，那就只好承认这里面很可能真的相关。当然，这里也许还有样本的数量多少，样本对于其他因素（例如地理因素、当时的社会政治因素等）分布的问题，需要做进一步的挖掘调查。但是如果数据表明真的相关，尽管暂时还不知道原因，那就得承认，进一步还可以根据这些相关性做出某些方面的预测并在实践中加以检验。事实上，后来有哥伦比亚大学医学中心的 Tatonetti 博士整理了 100 年中 175 万人的医疗记录，分析出生月份与 1000 多种疾病的关联，据称发现有五十几种疾病似乎是与出生月份有些关联的。不过，就目前而言，断言这些疾病确实与出生月份相关也还为时尚早。

还有个朋友在做的研究就更不一般了，他收集了大量的基因信息，试图从中挖掘基因排列的规律。基因的编码就像字符串一样，一个基因就类似于一个字符，然而这些“字符”的排列并非随机，那么这里面有些什么样的“词汇”？这些词汇的排列是否存在某种“语法”？这实际上也可以归结到相关性的问题。“词汇”取决于各种组合的出现频率，“语法”则取决于词汇之间的相关性。所以，这其实也是数据挖掘。

其实，从数据中特别是文字记载中发现相关性的思路和方法早已有之，并非现在才有。以历史研究为例，所谓考证、考据，其实就是在文字记载和历史遗迹中发现相关性的过程。但是从前的数据量太小，手段和设备太原始，跟现在的数据挖掘显然没法比。

1.3 并行计算

一般而言，数据量大了，所需的计算量自然也大。计算量与数据量之间的关系不一定是线性关系，一般是某种多项式的关系，甚至有可能是指数的关系。计算复杂性和算法分析就是专门研究各种问题和算法的计算量与数据量之间的关系。最理想的是常数关系，就是不管数据量有多大，计算量都固定不变，人们把这样的复杂性记为 $O(1)$ 。比方说，根据下标访问数组中的元素就是这样。理论上不管数组有多大，不管下标的值是什么，访问任何一个数组元素的开销都是一样的，与数组大小无关。可惜这样的算法是很少的。比这复杂一点的是 $O(\log_2 N)$ ，这里的 \log_2 是以 2 为底的对数，例如“对分搜索”的计算量就是这样。再复杂一点，就是线性复杂度，记为 $O(N)$ ，意为计算量跟数据量 N 成正比，这里的 N 表示数据的个数。再往上就是 $O(N\log_2 N)$ 、 $O(N^2)$ 等多项式复杂度。而指数复杂度，例如 $O(2^N)$ ，则对于大数据显然是不现实的了。但是，即使是对于线性复杂度的计算，由于大数据条件下的 N 很大，其计算量也是很大的。

当计算机的计算能力小于具体计算所需的计算量时，一个办法就是换用计算能力更强、计算更快的机器。在计算机技术发展的早期，人们就是这样做的，小型机不够快就用大型机，大型机不够快就用巨型机。但是，计算速度的提高毕竟赶不上数据量的增长，也赶不上复杂问题的涌现，于是人们就设法“分而治之”，一台机器算不过来就两台、三台、 N 台，使这些机器并行计算，希望 N 台机器合在一起可以使计算能力提高到 N 倍。另一方面，也在单台机器的系统结构上想办法，把机器做成“向量机”、SIMD（单指令多数据）、多核、多处理器等结构，总之是设法提高计算的并行度。并行度的增加就意味着综合计算能力的提高和计算速度的提高。

即便在单处理器的机器上，人们也设法通过多进程（或多线程）并发来增加并行度。读者也许会感到疑惑，在单处理器（且为单核）的机器上，一共就只有一个处理器在计算，多进程并

发怎么能提高并行度,怎么能提高计算速度?确实,并发不等于并行,一共只有一个处理器,就不会有处理器之间的并行了。但是,在单进程的系统中处理器常常会因为要等待输入、输出的完成而空转,等待的时间长度一般取决于外部设备。例如读磁盘文件的一个记录块,就可能会让处理器等上数十毫秒。可是,如果是多进程并发,那就可能把这数十毫秒的时间用于别的进程的运行。在磁盘驱动器忙于寻道,让磁头镇定下来,然后等磁道上的目标扇区转到磁头下面的过程中,CPU 在执行另一个进程的代码。数十毫秒的时间,已经可以做很多计算了。所以,此时的并行度是在 CPU 与外部设备之间。虽然一共才一个处理器,但是并行度确实还是增加了。

但是人们很快就发现,把机器或处理器的数量从 1 个提高到 K 个,一般而言并不能使计算能力也增加到 K 倍。这是因为,在这样的情况下需要把相当大的开销花在不同的机器或处理器之间的通信与同步上。更糟的是,许多问题其实很难找到合适的算法可以将其分解到不同的机器或处理器上去分头计算,然后再加以综合。所以,从前曾经有个猜想:把 K 个处理器合在一起,当 K 充分大时,综合的计算能力(速度)只能提高 $\log_2 K$ 倍。这就是说,假如我们把 256 个 CPU 连在一起,其综合的计算能力也许只能提高 8 倍。这当然是很令人沮丧的。这种现象特别多见于一些输入数据量不大而关系很复杂的问题。然而,如果虽然输入数据量很大,可是数据之间独立性很好,而且针对每一项数据所做的计算又很简单,那么 K 个处理器就真的能将综合的计算能力提高到 K 倍。所以,并行计算能否有效提高系统的计算能力,最终还是取决于数据的性质,取决于具体的问题和算法。

而 MapReduce 则正是这样一种并行计算的模型,它试图以 K 个处理器来分担对于 N 个数据的计算,希望能把总的计算时间降到 $1/K$ 。但是这个愿望只有在 N 个数据的计算互相独立,也即算法的复杂度为 $O(N)$ 时才能实现。在这种情况下,假设对 N 个数据的计算量(以计算时间表征)为 $aN+c$,那么 K 个处理器并行就有望将计算时间降至 $aN/K+c$ 。如果算法的复杂度高于 $O(N)$,则计算时间一般也能有所降低,只是不能按比例降低到 $1/K$ 了。所以,人们称 MapReduce 那样的并行计算为“极简并行”。

实践中我们可以看到两种不同的典型:一种是小数据大计算,即输入数据的量很小,但是计算量却很大、很复杂,这样的问题不适合于 MapReduce 那样的极简并行计算,需要为之搭建更复杂的数据流;另一种是大数据小计算,即输入数据的量很大,但是针对每项数据的计算却很小、很简单,而且这些计算互相独立,这样的问题就适合于 MapReduce。当然,还有一种可能是大数据大计算,即输入数据的量很大,而且针对每项数据的计算量也很大并且很复杂。这样的问题当然更不适合 MapReduce,更有待于算法上的突破。

1.4 数据流

在为增加并行度而发展起来的各种并行处理系统结构中,最引人注目的是“数据流(data flow)”结构。其原理是这样:考察数据从进入系统,经过各种处理,直到离开系统的整个流程,将此流程分解成若干阶段,其中的每个阶段都是一种相对独立的处理和计算,然后将这些阶段的计算分布在不同的机器节点上,不同节点之间只有数据的流通。这样,由于处理和计算的不同阶段之间,或者说上、下游之间(而不是数据样本之间)的并行度,就实现了另一种形式的并行处理。这好比一条河,在上游的水往下流的同时下游的水也在往下流,数据就像河水一样形成了数据流。这也很像工厂里的生产流水线,一个节点就是一个工位,数据就像流水线上的部件或半成品。

当年 Unix(以及现在 Linux)的管道(pipe)和流水线机制就体现了数据流的思想。我们不妨从一个 Unix 命令行说起:

```
cat shakespeare/*.txt | normalize | sort | uniq > vocabulary
```

这里的 cat、sort、uniq 都是由 Unix 提供的工具软件(utility), normalize 则是由程序员补充开发的一个工具软件,其功能是将输入的每个单词还原成它的原型,例如 works 和 worked 的原型是 work, did 和 done 的原型是 do, 等等。这个命令行把 4 个工具软件串在一起,启动后就建立起 4 个独立的进程。竖线“|”是由操作系统提供的“管道(pipe)”机制,将前面一个进程的输出传给下一个进程作为其输入。这样,第一个进程 cat 从文件系统逐个读出目录 shakespeare 下的所有.txt 文件,那是莎士比亚所有著作的字符文件,本来是要输出在显示屏上的,但是现在就由管道机制传到后一个进程 normalize 的输入端了。同样,normalize 的输出被传到 sort 的输入端,在那里进行排序,而 sort 的输出则被传到 uniq,消去重复出现的单词。最后 uniq 的输出被写入词汇表文件 vocabulary,这就是莎士比亚在其著作中用过的所有词汇。本来,如果我们要写一个从文本中提取词汇表的软件,这软件中对文本内容的处理也应该有这么几个阶段,而且这几个阶段各自所做的处理都有一定的代表性,都有重复使用的价值。事实上 cat 和 sort 都是很常用的工具。现在这样就可以把此类工具很灵活地加以组合,搭建出功能很强的流水作业线。因为进程之间流通的都是数据(没有控制信息,也没有地址信息),所以这是个数据流,其中的进程互相并发。当年 Unix 的成功在很大程度上来自这个新颖而强大的功能以及由此而来的编程风格,以至于“Unix 风格”成为一时的热门。

但是我们更感兴趣的是这里面的数据流思想。

先看对于这些进程的驱动。以 normalize 为例,它的操作是针对每个单词的,显然它只是在有输入数据到来时才工作,没有数据到来时就睡眠。换言之,这个进程是受输入数据驱动的。事实上,整个流水线,即整个数据流中所有的进程都是这样。所以,Unix 的这些工具性软件有个共同的结构模式:

```
while(e = next_element() != EOF) {  
    o = f(e); //对输入流中的每项数据进行处理,调用函数 f()。  
    output(o);  
}
```

每个进程都是在一个 while 循环中打转。每次都企图从输入端读取一项数据,然后加以处理,并输出本次处理的结果(如果有的话)。执行着不同工具软件的进程,其输入数据的单位可以是不一样的。如果是以字符为单位,那么这里的 next_element()就是 getchar();如果是以行为单位,那么 next_element()就是 getline();余可类推。注意:输入数据也可以是复合数据,例如 KV 对,即由一个键(key)及其值(value)构成的“键/值”对,例如“Name: Shakespeare”;甚至也可以是数据结构。不管是 getchar()还是 getline(),或者是别的什么 next_element(),都是只有在读到了数据的时候才返回,否则这个进程就会睡眠等待,让出 CPU,直到有数据到来时才被唤醒,这就是“数据驱动”,是数据流的一个重要特点。

再看数据在这些进程,或者说“节点”之间的流通。除输入数据的到来之外,各个进程之间并无别的同步机制,所以进展速度一定有快有慢。如果上游节点产生的输出数据快于下游进程对数据的耗用,那么数据就会在两个进程之间堆积起来。所以,每个进程的输入端一定得有

缓冲和排队的机制。Unix 的管道机制保证了这一点。

在上面这个例子中, `normalize` 这个节点对其输入数据是处理一个输出一个, 所以在节点内部不会有堆积。更重要的是这样有利于维持一个均匀的流, 就像河里的水, 一边流进来一边就流出去了。从这个角度看, 节点之间数据传输和处理的单位似乎是越小越好, 因为那样可以使数据流变得更平稳均匀。但是, 数据的传输是有开销的, 如果单位太小了, 开销所占的比重就会上升, 那就不划算了。所以数据传输的单位大小需要加以权衡。

另一个节点 `sort` 就不同了。我们知道, 排序是不能让数据随到随走的, 必须等所有数据都到位之后才能进行操作。所以, 数据会在 `sort` 这个节点上堆积起来, 直至所有数据都流入这个节点之后才会有输出。所以, 像 `sort` 这样的操作就像一座水坝一样, 对数据流有破坏作用, 这个节点与下个节点之间的数据流变成了工作流。所以, 只要一个流程中含有排序操作, 这个流程就天然带有“批处理”的特征。不过, 尽管如此, 流的形状或拓扑却保持不变, 所以我们常常不加细分而仍旧称之为数据流。

再看并行度。Unix 的流水线是建在单机上的, 进程之间是并发而不是并行的关系。如前所述, 并发确实也会带来一点并行度, 但是毕竟有限。要有更大的并行度, 就得变并发为并行, 办法之一就是把 `cat`、`normalize`、`sort`、`uniq` 放在不同的机器上, 形成一个数据流机群。画成拓扑图如图 1-1 所示。

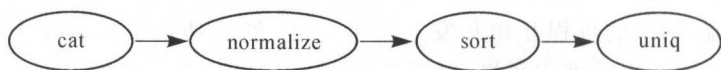


图 1-1 单机数据流拓扑图

在图 1-1 中, 节点之间的边都是有向的, 代表着数据的流向, 所以是个有向图。另外, 这个图中并没有形成环, 节点之间并无反馈。这样的图, 就称为“有向无环图”, 缩写为 DAG。不过, 这只是一个链状的一维 DAG, 也即链状的一维数据流。这样的数据流当然是简单的数据流。复杂一点的会有分叉而成为树状, 那就是二维的数据流、二维的 DAG 了。再复杂一点的数据流还会有反馈, 那就不是“无环”, 不称其为 DAG 了。

Unix 的管道机制并未提供分叉的功能, 它确实提供了一个工具软件 `tee`, 意为可以用作 T 形节点, 但是 `tee` 的输出只能到文件, 而不是到 `stdout`, 也不允许重定向 `tee` 的输出。所以 Unix 在单机上只允许构建链状数据流。

显然, 把这些节点放在不同的机器上就获得了真正的并行度, 这种并行度存在于上下游节点之间。在这个例子中, 开始时前面三个节点可以并行, 只有 `uniq` 暂时空着; 后来前面三个节点都完成了操作, 这时候就只有 `uniq` 在忙了。姑且假定每个节点的计算量都是 $1/4$, 用于数据传输的计算量忽略不计, 并且耗费的时间完全取决于计算量, 那么原来完全串行时的总时间是 1, 现在则降到了 $1/2$ 略多。如果不是 `sort` 把数据流变成了工作流, 那么速度还可以提高。

可是这里只有上下游节点之间的并行, 也即前后工位的并行, 却没有同一工位内部, 即数据之间的并行。如 `normalize` 就始终只有一个节点在工作, 当数据量很大时显然我们也需要在这里引入并行。于是我们可能就有如图 1-2 所示的一个拓扑图。

这里配备了三个 `normalize` 节点。这样一来, 原来一维的链就成了二维的图。不过我们也可以仍把它想象成一维的链, 只是在 `normalize` 节点的位置上是三个叠在一起, 就像三张相同的透明胶片叠在一起, 我们仍旧只看到一个。或者也可以说, 这个图的投影仍是一维的链、

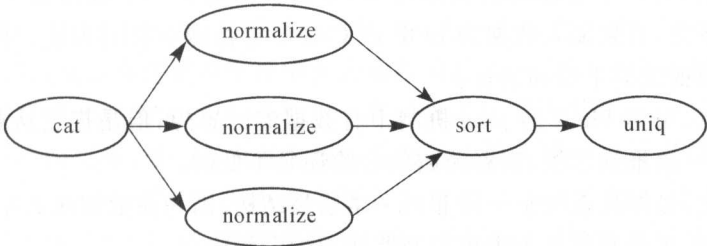


图 1-2 三个 normalize 节点数据流的拓扑图

一维的 DAG。

但是 cat 仍只有一个,这是因为从磁盘读出的速度总是比逐个单词处理的速度快,因而一个 cat 节点的读出就足够三个 normalize 节点之用了。图 1-2 中 cat 的输出被分发给三个 normalize 节点,严格说来相当于在 cat 后面还有一个实现负载均衡的 distribute 节点。

值得指出的是,sort 也只有一个,这就不合理了。按理说,sort 的计算量比 normalize 还大,理应有更高的并行度才对,为什么反倒只有一个呢?这显然是应该改进的。问题是 sort 这种计算不太容易并行化,比方说“冒泡排序”,就难以并行化。幸好还有一种“合并排序(merge-sort)”的算法,可以先分块进行局部排序,然后再加以合并。于是我们就把这个数据流的拓扑变成图 1-3 所示的那样。

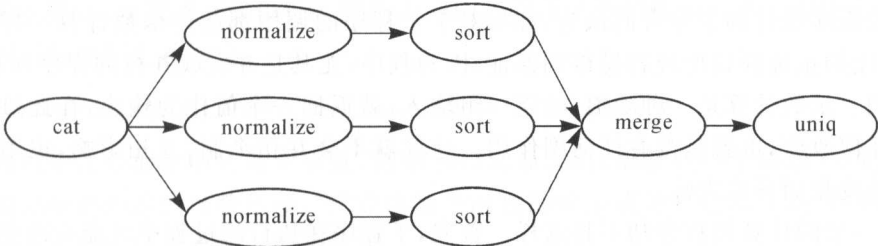


图 1-3 合并排序数据流的拓扑图

这样明显合理得多,性能也肯定会好很多。这个数据流的拓扑看似是个 DAG,但仍属于很简单的 DAG,本质上仍是一维的链状 DAG。

这里的每个节点都是运行着某种软件的处理器,但也完全可以是一种专用的硬件芯片或器件。从这个意义上说,我们可以把收音机、电视机等也看成是数据流系统。作为一个数据流系统,电视机的输入来自天线,里面经过调谐、放大、检波等节点,然后就分叉成视频和音频两个分支,各自又经过种种处理,最后视频分支的输出投射到显示屏,音频分支的输出则用来驱动扬声器。这整个拓扑画下来就是个 DAG,只是每个节点都是专用芯片而不是处理器,而且在节点间流通的主要是模拟信号,而不是数字化了的信息(现在有数字电视了)。所以,数据流在某种意义上是对电子线路的模仿。另外,像电视机那样,从某一节点的输出开始就分成音频和视频两个不同的支流,才是实质上的二维数据流。

仍以电视机进行类比。电视机的运转与电视机的装配是两码事,装配归装配,运转归运转。数据流系统也体现和遵循着同样的思路。以前面的 Shell 命令行为例:Unix 为使用者提供了 cat、sort、uniq 等“元器件”,因为不够就自己开发一个 normalize;同时 Unix 又为使用者

提供了 Shell 及其语言。于是使用者就运用 shell 语言搭建了一个数据流。一旦搭建起来之后它就一直运转下去,直到输入数据为 EOF 或 Ctrl-C 等特殊字符时为止。但是 Unix 并没有提供跨机器搭建数据流的手段和语言。

综上所述可见,虽然 Unix 的 pipe 机制中有数据流的思想,但是把它从并发变成并行,从单机推到多机,从一维推到二维,需要跨越的台阶还是不低的。

最后还要指出,数据流系统是一种非冯·诺依曼结构,因为在数据流的拓扑中不存在一个全局的中央控制器,更没有供这个中央控制器使用的全局内存。

1.5 函数式程序设计与 Lambda 演算

数据流主要是搞系统结构的人发展起来的概念和技术,这方面的鼻祖之一是 MIT 的 Jack Dennis,那还是 20 世纪 60 年代中期的事。它最初是作为异步控制模块的电路结构开始的,到 1975 年前后就已经在讨论数据流语言和数据流计算机了。但是一些搞软件和计算的人也与他们殊途同归,发展起一套称为“函数式程序设计(functional programming)”的理论,其虽然形式上不同,但实质上却是与数据流等价的。

函数式程序设计最初是作为保证程序正确性的一项措施而发起的,但是它的根源却可以追溯到 20 世纪 30 年代丘奇(曾是图灵的导师)等人对于计算理论的研究,特别是关于“ λ 演算(Lambda calculus)”的研究。

函数式程序设计的倡导者们认为,如果对于子程序的调用都能像函数计算一样没有副作用,那么程序的正确性就比较容易得到保证,因为程序(尤其是并发或并行的程序)的毛病往往出在副作用。函数计算是一种映射,给定一组输入,就返回一个值作为输出,在此过程中对于环境没有任何改变,也就是没有任何副作用。这样就不会互相牵制、互相影响,也不会因为有意外的环境变化而导致失败。

但是,一般的计算机程序却不是这样。首先,子程序在执行的过程中可能会改变全局量的值。还有,如果作为调用参数的变量是指针,或是实质上相当于指针的变量引用(reference),那么其所指向的变量就可能改变。这些都发生在函数返回值以外,所以称为副作用。要确保无副作用,首先当然不能允许有全局量存在(Java 就不允许,但是如果可以通过 get()、set()之类的方法引用别的对象中的变量,则实质上与全局量无异),同时还得保证函数调用时只能传变量的值,而不能传地址或变量引用(实质上就是地址)。也就是说必须是“by value”而不能是“by reference”。另外还有一种表述是:调用参数所涉及的变量(对象),必须是“immutable”,即“不可更改”的。我们举个例子来说明。假定有一个字符串,在一个缓冲区 s 中,然后我们需要提供一个将字符串转换成大写的子程序。一般我们会怎么做呢?可能是这样:把缓冲区 s 作为参数传给一个子程序 to_upper(),这个子程序扫描作为参数传下来的字符串,就地将这个字符串中的每个字符改成大写,直到行的末尾。这样,当子程序返回时,这个字符串就成了大写的字符串。这是典型的“mutable”(可更改),一个原先就存在的字符串的内容被更改了。如果另外有一段程序预期这个字符串的内容是原始的内容,那么在 to_upper()执行之前这个预期是成立的,但是在 to_upper()执行之后就不成立了。这样,那段程序的运行结果就与时序有关,而这可能不是原先所考虑和预期的。那么“immutable”的调用是怎么做的呢?调用者会先复制一个副本,然后把这个副本作为参数传给子程序(这就是“by value”)。

或者,虽然仍是把字符串的原件传给子程序,但是却将原件设置成“只读”,不让更改(这得要有措施保证),这样,就不会有副作用了。而在子程序中,则另外分配一个缓冲区,把这个字符串拷贝过去并改成大写,最后将这新的字符串作为函数值返回,这样就不会有副作用了。

结合函数式程序设计的理论,人们开发出一些函数式程序设计语言,一方面从语言上保证函数式程序设计原理的实施,另一方面也为程序员进行函数式程序设计提供了方便。这些语言都是“陈述式(declarative)”的,而不是“指令式(imperative)”的。什么意思呢?所谓“declarative”,就是只要说明哪一个参数或变量(一定是局部变量)的值是来自哪一个函数的就行,并不需要像“imperative”的程序那样去给出一步步的操作指令(语句)。这就带来函数式程序设计的一个特点:程序的语句次序是无关紧要的,前后换一下也没有关系。之所以如此是因为:对于函数式程序设计,我们只需陈述函数间的调用关系,而执行时的操作顺序则自然地隐含于这些调用关系之间。设想,如果函数 $a()$ 的输入来自 $b()$ 的输出,而 $b()$ 的输入来自 $c()$ 的输出,那么对这两句话来说,你先说后面那句、后说前面那句意义是一样。实际上,这样的程序已经不能算是“程序”了,因为它并没有规定一步步的“程”和“序”。不过这种陈述式的风格未必能坚持到底,到了底层的函数内部,有时还是需要有点指令式程序的。那也不要紧,反正语言通过其编译器或解释器保证了无副作用。注意:一般 CPU 的指令系统都是指令式的,函数式程序设计语言的编译器会把陈述式的语句翻译成指令式的程序。实际上,陈述式的“程序”所表达的是怎样搭建数据流,就好比是电视机的电路图。按电路图装配电视机的时候,先装上这个元件还是那个元件没什么关系,反正把电路图中所有的元器件都装上就可以了。

令人惊讶的是,函数式程序设计与数据流结构之间竟是如此贴合一致。为什么这么说呢?我们不妨回到前面那个数据流:

```
cat shakespeare/*.txt | normalize | sort | uniq > vocabulary
```

如果我们把这数据流中的每个节点,例如 `cat`,又如 `sort`,都看成一个函数,并把节点之间的管道看成函数调用时的参数传递,“|”左边那个函数的输出成为右边那个函数的输入参数,那就可以把这个数据流写成如下的函数式:

```
vocabulary = uniq(sort(normalize(cat(shakespeare/*.txt))))
```

就是说,`cat()`的输出作为 `normalize()`的调用参数,而 `normalize()`的输出又作为 `sort()`的调用参数,余类推。

那么把数据流中的节点看成函数有没有道理呢?有。因为函数无非就是映射,把定义域上的一组参数映射到值域上的某一个点上;而数据流中的节点,则根据一组输入参数计算出一个输出值。因此,我们可以认为后者是前者的实现。而且,输入到节点参数都是数据而不是地址,是“by value”;而任何一个节点都无法直接改变别的节点中的变量,所以那些变量都是“immutable”的;最后,这些节点没有共享变量,更没有全局变量,所以一个节点对一组输入数据的计算相当于一次无副作用的函数调用或函数计算。

因此,我们可以把一个数据流所进行的计算看成是对一个复合函数的循环调用;反过来,如果我们有一个复合函数式,那么把它实现出来就应该是一个数据流。

上面所涉及的几个函数都只有单个调用参数,而且函数的输出也只出现在一个函数的输入参数表中,这决定了与其对应的数据流乃是链状的数据流。可想而知,如果一个函数有多个输入参数,那么数据流中与这个函数对应的节点就起着汇聚的作用。而倘若要让数据流在某

一节点上分叉,那就得把这函数的输出赋给一个临时变量,再让这个临时变量出现在多个函数的输入参数表中。

不过要是直接用上面这个函数式作为程序中的语句,则有个缺点:其形式上的次序与对应的数据流是反的,这有点不合一般人的思维习惯。而如果用面向对象的语言来表达,就可以写成例如下面这样的语句:

```
vocabulary = cat(shakespeare/*.txt).normalize().sort().uniq()
```

这里的语法和语义是这样:函数 `cat()` 的返回值是个某类型(比方说 `Words`)的对象,输出的内容(字符串)就存在这个对象(数据结构)的内部变量(缓冲区)中。此外,这个类提供一个方法函数 `normalize()`,其输入取自该对象内部的缓冲区,这个函数也返回一个某类型对象。这个某类型与前面的某类型可以是同一个类型,也可以是不同的类型,但是原理和模式是一样的:函数 `normalize()` 的输出写在该类对象的某个内部变量(缓冲区)中,这个类则提供一个方法函数 `sort()`。此后可以类推,这个语句可以一直延伸下去。但是请注意,即使 `normalize()` 所返回的类型与其前面的 `cat()` 所返回的类型一样,即 `normalize()` 的输出与输入是同一个类型,却不是同一个对象,而只能是同一类型的两个不同实例,因为输入参数的内容是“immutable”(不可更改)的。可想而知,如果数据流中的节点分处于不同的机器,中间通过网络相连,那自然是满足这个条件的,因为下游节点接收到的数据只能是上游节点输出的一个副本,下游节点的输出不可能回过头去改变上游节点的输出。即使是在同一台机器上,只要进程间不共享内存而是通过报文传递(message passing)实现通信,那也自然是能满足这个条件的。

但是这里还有个问题。无论从上面的函数式看,还是从与之对应的程序语句看,其中对于函数的调用都像是单次的调用,如 `normalize()`、`uniq()`,似乎就是单次的函数计算,可是实际上这些函数的输入都是序列,相同的函数计算要施加到输入序列中的每一个元素上。这就是为什么在 Unix 上,这些 utility,即工具性软件内部的操作都是套在一个 while 循环中的。像这样依次施加在输入序列中每一个元素上的函数计算,在当年丘奇的理论中,就称为 λ 演算,即“Lambda calculus”。

所以,在 Unix 的 shell 命令行中,以“|”号分隔的程序名相当于数据流中各个节点的操作名,从而也相当于函数名,并且带有 Lambda 语义。换言之,作为脚本语言的 shell,是带有 Lambda 语义的。可是,在 Java 一类的编程语言中呢?如果我们一如既往地吧函数调用看成调用一次执行一次,那么要表述 Lambda 演算就得把 while 循环写进去。那样的话,要用函数式的语句来描述数据流就有困难了。反过来,要是我们把函数调用看成是 Lambda 演算,那么真要表达单次的函数计算时又有问题了。这提示我们,对于同一个函数,语言中对于函数调用应该有两种不同的语法,分别表示两种不同的语义。进一步,用来实现 Lambda 语义的 while 循环不宜放在具体函数的内部,而应该放在外面。但是,不应该要求程序员自己编写这个 while 循环,也不应该让这个循环显式地出现在程序中,而应该由编译器自动产生。传统的指令式编程语言大多忽视 Lambda 演算,这是因为从前的计算不太有这方面的要求(当然也不绝对,要不然 shell 就不会实现这样的语义了)。但是现在情况不同了,大数据的出现将 Lambda 演算的重要性提高到了前所未有的高度。

正是因为这样,Java 语言才有了它新的版本——Java 8。较之老版本,Java 8 主要的改进就是两个方面的扩充:一是增添了 Lambda 演算的语法和语义;二是提供了一个实现“流

(Stream)”式计算的 API。这里所说的“流”就是数据流。

将 Lambda 演算作为语言成分加进 Java 8,意味着 Java 8 的编译器能解析按规定语法书写的 Lambda 演算语句,并能按规定的语义生成为此所需的程序结构,包括上述的 while 循环,以及在节点间传递数据的机制。而在此之前,当 Java 语言还不提供这样的语法和语义的时候,则需要有个“平台(platform)”或“框架(framework)”来提供这样的程序结构,而实际进行计算的函数,结构上也会因此而有所不同。实际上,Hadoop 在某种意义上就是这样的平台和框架,而 Spark 则在 Java 尚未提供函数式程序设计的语法和语义的时候就采用了函数式的 Scala 语言。

不过编程语言中有了 Lambda 和 Stream 的语义和语法也不是就万事大吉了。其实编译器或解释器所生成的代码也是建立在系统底层的基础上,依靠底层支撑的。要是早期 Unix 没有提供 pipe 机制(当时还没有 socket 这些机制),则 Shell 语言的数据流语义就根本无法实现。另一方面,Shell 虽然提供了数据流的语法(用“|”代表管道)和语义,但具体的实现只是在单机上的,数据流中的节点都只是同一台机器上的独立进程,对于多机的环境它就不能支持了。再说,如前所述,在 Shell 搭建的数据流中一个节点只是一个并发进程,而没有更高的并行度。所以,即使 Java 8 有了这方面的语法和语义,也有如何从单机推广到多机、如何增加并行度的问题。一般这可以通过不同的(底层)库程序(library)加以实现。

至此,我们已经能够理解,函数式程序中的语句本来就不必像传统指令式程序中那样去一步一步细述如何操作,而且语句的次序也变得无关紧要,因为操作的次序已经蕴含在函数表达式即程序语句之中了。函数式程序就好比数据流的装配图,先装这个还是先装那个没有什么关系,只要按图施工把它装配完成,就可以开动运行了。而函数式程序设计语言的编译器(或解释器),就起着“施工单位”的作用。细想 Unix 上的 Shell,对于前面所述的那个命令行来说,不就是起着把 cat、normalize、sort、uniq 这些进程通过 pipe 装配在一起的作用吗?而 Unix 这个操作系统,则为此提供诸如 pipe()、dup()、fork()等系统调用,使 Shell 能够有这样的装配能力。不过 Unix 上的 Shell 是在单机上装配的,而多机(集群)条件下则需要有跨机器的装配能力。

1.6 MapReduce

数据流的概念和技术,虽然在程序的模块化和可重用(可重复使用)方面也有很大的意义,但主要还是由于提高并行度的要求而出现的,是并行处理的一种重要方法。前面讲过,如果考虑作为输入数据的样本数量和所进行的计算操作这两个因素,有些问题是小数据大计算,有些则是大数据小计算。当然,还有小数据小计算和大数据大计算两种可能,前者不足挂齿,后者则当然更具挑战性。但是,对于大数据,“化整为零,分而治之”的策略总是不错的,更何况大数据中的数据本来就可能是“零”的而不是“整”的。但是一般而言,“分”也不是一分了之,最后还得汇总。所以,应该是先分头处理,然后对结果加以整合。按这样的思路设计的数据流模型,就称为 MapReduce,意为先 Map 然后 Reduce,其结构和拓扑如图 1-4 所示。

在这种模型中,输入数据被分成 N 份,分别由 N 个实施 Map 计算的 Mapper 节点分头加以处理,处理的结果由 M 个实施 Reduce 计算的 Reducer 节点加以汇总。在实际使用中, M 常常是 1,即只有一个节点进行 Reduce 计算。什么叫 Map 计算呢?实际上就是某种函数的 Lambda 计算。Map 这个词,一方面来自 Lambda 计算的理论,另一方面函数的本意就是映射。一般而言,Map 计算不一定显著减少数据的数量,往往是有多少输入就有多少输出(但是

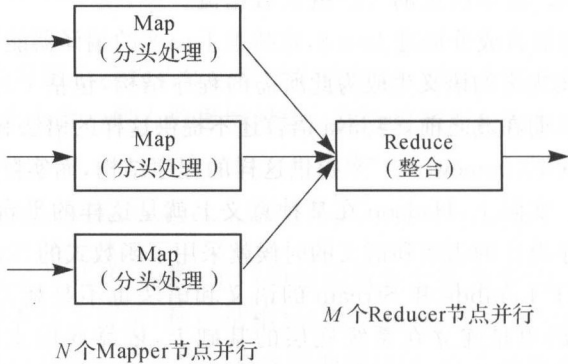


图 1-4 MapReduce 拓扑图

内容变了)。而 Reduce 计算则是整合和汇总,其一般会把数据的数量大大减少下来,所以叫 Reduce。可见,这里的假设是数据互相独立,并行度仅存在于数据(样本)之间。

这样的并行处理实在是太简单了,这样的办法谁都能想到,简直是简易得令人不好意思提起,以至于人们称之为“embarrassingly parallel”,意为简单到令人尴尬的并行处理。不过,模型固然简单,但要真正把它实现出来也并非易事。

既然 MapReduce 这个模型很简单,当然不会是到了最近才有人想起而突然发明出来的,而是早已有之。但是为什么它一直悄无声息,到现在才一下子大热起来呢?因为以前人们想用并行处理解决的大多是小数据大计算的问题,那些问题的输入数据量并不很大,但是算法却相当复杂,或者数据之间互相牵连,少有能将它们分头加以处理的算法,总之是都不太适合用 MapReduce 的模型加以并行处理的。而有些适合用 MapReduce 解决的问题,则数据量还不小,那时候也还没有“精准营销”一类的概念和要求。是互联网的出现和发展使情况发生了变化,因互联网而来的许多活动产生了大量的数据,而且需要对每份数据进行的计算大多并不复杂,互相的牵连往往也不多,从而就催生了许多属于大数据小计算且适合用 MapReduce 模型加以解决的问题。再说,计算机技术的发展也使得采用大规模集群进行计算的成本大大降低而变得可行。近年来大数据技术,特别是 MapReduce 技术成为热门技术,正是这些因素综合作用的结果。

在 MapReduce 模型中,Map 阶段的并行度一般是比较高的,Mapper 节点的数量 N 远大于 Reducer 节点的数量 M 。这样,Map 阶段的并行度至少就是 N ,而整体上是否能比 N 更高则取决于 Map 和 Reduce 这两个阶段之间是否有并行,Reducer 节点在 Map 阶段是否空闲。如果 Mapper 与 Reducer 之间是数据流的关系,即 Mapper 节点一边处理一边就把结果发送给 Reducer,使其可以同时进行处理,那么 Map 阶段的并行度甚至可以达到 $N+M$ 。而如果 Mapper 与 Reducer 之间是工作流的关系,则 M 个 Reducer 节点在 Map 阶段只能等待,并行度就会下降一些。

Map 阶段结束之后,如果 Mapper 与 Reducer 之间是个均匀的数据流,而且节点负载得到均衡,那就没有明显的 Reducer 阶段了,因为在 Map 计算的同时 Reduce 计算也在进行,到 Map 阶段结束的时候 Reduce 也基本完成了。但是,如果两种计算的负载达不到均衡,或者 Mapper 与 Reducer 之间干脆是工作流的关系,那么 Map 阶段结束之后还会有一个 Reduce 阶段,此时的并行度为 M ,因为 N 个 Mapper 节点此时都使不上劲了。

最后还要再次强调指出,并非所有的大数据问题都适合用 MapReduce 解决。

1.7 大数据处理平台

因为历史的原因,“系统”、“平台”、“环境”这些词的含义从一开始就有点模糊。按理说,“系统”应该是一整套提供着某种特定应用功能的独立而完整的设施,里面包括具体的应用和支撑这些应用运行的“平台”和“环境”。而“平台(platform)”,则是为各种具体应用提供用来构成系统的基础设施。比方说,汽车的底盘就叫 platform,在上面加上不同的车厢和发动机,就成了不同车型的汽车,那就相当于一个系统。可是,按这样的理解,“操作系统”就应该称为“操作平台”,可“操作系统”却是早就已经约定俗成的称呼。再说,操作平台也并不等于操作系统,因为在应用和操作系统之间还有各种中间层。

就大数据处理而言,由于数据量大导致计算量大,就需要有多机(集群)并行处理。既然如此,具体的应用软件(比方说数据挖掘)就不能直接以单机操作系统为平台,而需要有一个面向计算机集群、支持并行计算特别是数据流计算的跨机器节点的统一平台。在某种意义上,这样的大数据处理平台就相当于“集群操作系统”。

不过,就像单机操作系统,即单机上的处理平台一样,应用的目标不同,大数据处理平台的结构和功能也就应该有所不同。平台是为应用服务的,应该适应具体应用的要求,而不是让平台决定可以有什么样的应用。那么针对不同的应用目标会有些什么样的大数据处理平台呢?

首先是能支持什么样的数据流拓扑。有些应用只要有极简的 MapReduce 就够了;还有些应用则要求能支持链状数据流,有些则进而要求能支持一般的 DAG,即树形的数据流;有些可能还要求允许有成环的数据流拓扑。

即使数据流的拓扑相同,是否支持实时处理也是个重要的需考虑的问题。如前所述,具体的应用有 OLTP 与 OLAP 之分。前者,即“在线事务处理”,是有实时要求的,这样的处理必须是真正的数据流(而不是工作流)才能支持。而后者,即“在线分析处理”甚至“离线分析处理”,则“批处理”式的工作流就可以支持。

应用界面上为用户提供什么样的语言,有什么样的语法语义,也是个问题。

还有系统可靠性和安全性的考虑和措施。

到了具体实现的层面,则还有更多的因素。

所有这些因素的不同组合,就决定了可以有结构上和形态上不同的各种大数据处理平台。而本书所介绍的 Hadoop,只是各种可能中的一种,但也是被广泛接收和使用的一种。

1.8 Hadoop 的由来和发展

2004 年前后,Google 公开发表了三篇论文,报道了 Google 内部正在使用的三项大数据处理技术。其中一篇是关于 MapReduce 的,一篇是关于面向集群的分布式文件系统的,还有一篇介绍了一种称为 BigTable 的分布式非 SQL 数据库。三篇论文围绕着一个中心,就是如何在一个由大量普通商品计算机(特别是价格不贵的 PC 机)构成的集群上进行大数据处理。在此之前,人们普遍认为大数据处理一定得有价格昂贵的大型机和专用机才能胜任,PC 一类的普通商品计算机是用不上的。

这三篇论文引起了广泛的注意和兴趣。因为 Google 是互联网应用的龙头企业,在实际运

营中积累了海量的数据,对于什么样的技术可以或比较适合用于这方面的大数据处理,它自然是“春江水暖鸭先知”。特别是关于 MapReduce 的那篇论文,使人们了解到:原来被认为是简单到令人尴尬的“embarrassingly parallel”或曰“极简并行”,在互联网相关的大数据处理方面竟已大有用武之地,已经可以用来解决许多问题。换言之,对于互联网相关的大数据处理,在相当程度上有 MapReduce 就可以应付了。

于是很快就有了立足于前两篇论文的大数据处理平台开源项目,这就是 Hadoop。至于第三篇论文,则有了基于 Hadoop 平台的分布式非 SQL 数据库开源项目 HBase,意为 Hadoop 上的数据库。早期的 Hadoop,直到 2.0 版之前,在数据流方面仅仅支持 MapReduce 的拓扑和结构,但是 2.0 版之后已经开始多样化。

Hadoop 是一种主要面向 MapReduce 的大数据处理平台,或者说是一个面向 MapReduce 的大数据处理系统的框架,这个框架中什么都有,就剩下针对具体应用的 Mapper 和 Reducer 两个模块的位置留待用户自己提供,这两个模块必须是按规定 interface 实现的 Java 类。把用户开发的 Mapper 和 Reducer 嵌入这个框架,就构成一个针对具体应用的、基于 MapReduce 的大数据处理系统。具体的 Mapper 和 Reducer 尽可以换,Mapper 和 Reducer 的数量即 Map 阶段和 Reducer 阶段的并行度也可以分别指定,但是数据流的拓扑是固定的,那就是两个节点的链状数据流,这是由框架的结构决定的。

另一方面,Hadoop 是面向计算机集群的。虽然在单机上也可以运行,但是 Hadoop 的设计目标是针对计算机集群的(要不然就不用这么复杂了),集群的规模可大可小,小则三五台,大则数千台甚至更多。整个集群连成一个局域网,各节点机上的操作系统一般都是 Linux, Hadoop 就架设在 Linux 上面(Hadoop 也有 Windows 版的),把这些节点机连在一起构成统一的处理平台。

除 MapReduce 框架之外,Hadoop 还提供了一个容错的分布式文件系统 HDFS,这是 Hadoop 除 MapReduce 之外的另一个子系统。显然这两个子系统分别对应着 Google 的两篇论文。HDFS 是以各宿主机的文件系统为基础、为支撑的,但是宿主机的文件系统只是本地的、局部的,而 HDFS 是全局的,HDFS 的一个“记录块”对于宿主文件系统而言就是一个文件。HDFS 文件以记录块为单位分布在许多节点上,在对文件中的数据进行处理的时候,数据存放在哪里,计算(Map 计算)就在哪里进行,而不是把数据远程调运过来供计算之用。当然,这需要有 HDFS 与 MapReduce 两个子系统之间的协调。

这样,单机操作系统如 Linux 是 Hadoop 的宿主,而 Hadoop 是对宿主的延伸和扩充。所以某种意义上 Hadoop 是对单机操作系统的延伸,实质上就是一个集群操作系统。

在 Hadoop 的发展历程中,2.0 版的推出有着里程碑的意义。首先,此前的 Hadoop 只支持 MapReduce 一种模式、一种拓扑,并且其 Mapper 和 Reducer 必须是用 Java 语言写成的类(class);而 2.0 版之后则也可以支持别的链状拓扑,并且不再限于使用 Java 类。更重要的改变是,Hadoop 的 2.0 版引入了一种新的作业管理机制,称为 YARN,将作业管理的权力下放,把原先集中在主节点上的作业管理分布到基层,使集群中各节点的负载变得更均匀,也使集群的运行变得更健壮。

Hadoop 的程序是用 Java 语言编写的。应该说这不失为一个正确的选择,因为 Java 的执行效率虽然是比 C/C++ 低,但是却以此为代价换来了许多好处。其中最重要的是 Java 语言显著降低了编程的难度,显著降低了对于程序员技术水平要求,从而能显著加快程序员们的

研发进度并显著提高“成品率”。不妨举个例子。在需要连续运行的那些软件的开发中,内存管理是个颇为头疼的问题,一不小心就会有内存泄漏,就是软件运行时分配到了内存,但使用后没有归还。有内存泄漏的软件,如果在使用中只是短暂运行,或小规模运行,那倒也不至于有问题,很可能不会被察觉,因为程序的运行没等内存泄漏到很严重的地步就完成了,程序一结束,进程一退出运行,就一了百了,把问题掩盖过去了,操作系统会处理进程的善后事宜,把所占的内存全部释放。但是如果是长时间运行,或大规模运行(海量的数据需要处理),内存的泄漏问题就“纸包不住火”了,哪怕很小的泄漏也会很快积累起来。但是,如果采用 Java 语言编辑,这问题就不用程序员操心了,因为 Java 语言的废料回收(garbage collection)机制自会处理释放内存的问题。这就使程序员省心了,对程序员技术水平的要求当然也就降低了。除内存管理之外,后面我们将看到,Java 语言中的“反射(reflect)”、“标注(annotation)”等,也是 C/C++ 不提供但却很有用的机制。总的来说,C/C++ 是面向精英的,水平高超的程序员可以用 C/C++ 写出十分精炼而高效的程序,但其入门的门槛却比较高。而 Java 是面向普通程序员的,难度相对较小,不过程序的执行效率和空间效率都要低一些。在计算机硬件的性能规格大大上升而价格大大下降、劳动力紧缺并且成本上升后的今天,以牺牲一些性能为代价换取程序的可靠性并降低对程序员的技术能力要求,显然是合理的、有利的。

1.9 Hadoop 的 MapReduce 计算框架

如上所述,Hadoop 是个面向 MapReduce 的大数据处理平台,为 MapReduce 计算提供了一个框架。这是什么意思呢?打个比方:Hadoop 就好像一块印刷电路板,上面什么都连好了,就留出两个芯片插座,一个用来插 Mapper 芯片,一个用来插 Reducer 芯片。使用者自带这两种芯片,在电路板上插上你的 Mapper 和 Reducer,再设置好参数,即 Mapper 要复制几份、Reducer 要复制几份,然后一按电钮,机器就运转了。这个假想的电路板就起着“框架”的作用。框架是固定的,而 Mapper 和 Reducer 则是可更换的。

也就是说,数据流的拓扑是在框架中固定连接好的,那就是链状的 MapReduce,但是采用什么具体的 Mapper 和 Reducer 则留待实际使用的时候才确定,而且各自的并行度可以设置。用前面打过的一个比方就是:Mapper 和 Reducer 都画在透明胶片上,在 Mapper 的位置上可以是 N 块胶片叠合,Reducer 的位置上可以是 M 块胶片叠合,但是我们眼睛所见的就只有两个节点,就是 Mapper 和 Reducer。

不过,使用者虽然可以指定有几份 Mapper 和几份 Reducer,却不能严格指定将它们放在哪些节点上,Hadoop 会根据输入数据所在的地点和相关节点的负载情况自动加以安排。

使用者自带的“芯片”可以是外购的,也可以是自己开发的。Hadoop 为此提供了一个 Mapper 类和一个 Reducer 类,插到这框架上的必须是对这两个类的继承和扩充。

说是两个类,其实起关键作用的就是两个方法函数,即 `map()` 和 `reduce()`,这就是系统应用层的核心所在。这两个函数所做的计算,按理说都是 Lambda 演算,不过在函数中不需要像在 Unix 的那些 utility 中一样写上 while 循环。当然,既然是 Lambda 演算,while 循环还必须要,但那是由框架提供的,Mapper 节点和 Reduce 节点上的 Hadoop 框架会各自循环调用 `map()` 和 `reduce()`。另外,数据流中每个节点的输入端还必须有一个队列,用来吸收上下游节点之间的速度差异和波动。在 MapReduce 这个模型中,其实只有 Reduce 节点才有这个需要,

所以 Hadoop 的 MapReduce 框架也为 Reduce 节点提供输入队列。

数据流的关键之一是数据怎么在节点之间流通。从逻辑上看, MapReduce 这个模型中的数据流通都是单向的, 都是从 Mapper 的输出端流到 Reducer 的输入端(要不然就不叫有向无环图了)。不过这只是就应用层而言, 在传输层和网络层当然会有双向的握手, Hadoop 会在 Mapper 与 Reducer 之间建立起 TCP 连接, 上下游节点之间在传输层以 TCP 报文为单位, 在网络层则以 IP 包为单位传输数据。

下一个问题是权衡数据传输的粒度, 粒度太小则开销太大, 粒度太大则上下游的并行度可能会降低, 且实时性变差。进一步, 如果上下游之间有排序, 则如前所述数据流将变成工作流, 那就变成了批处理, 完全丧失了实时性。可是排序对于许多 MapReduce 计算也确实是需要的, 即使为此而丧失了实时性, 对于许多大数据处理的工作而言很可能也还是值得的。OLAP(在线分析处理)不同于 OLTP(在线业务处理), 对 OLAP 的计算本来就并无实时要求, 并且往往要把同一批输入数据翻来覆去地梳理挖掘好多遍, 而 Hadoop 的设计目标恰恰就是用于 OLAP。至于并行度的降低, 则由于 Mapper 节点与 Reducer 节点的数量之比常常会很大, 所以在 map 阶段让 Reducer 节点空闲也没有多大影响, 而 Reducer 阶段的计算则通常都很简单(只是汇总一下), 应该不会占很长时间。

这样权衡下来, Hadoop 的设计者选择了在框架中自动提供对于 Mapper 输出数据的排序。其实 Hadoop 在这个问题上还是为用户提供了一定的灵活性, 框架上的这一大块部件(软件)是可以被替换的, 只是用户需要自己开发这样的替代品, 实际上好像也没听说有人这样做。

Hadoop 平台运行在集群上, 但一般并不是举整个集群之力来进行仅仅一个 MapReduce 作业的计算, 而是像在分时系统上那样, 可以同时也在集群上进行多个作业的计算, 这些作业之间既可能是并行的, 也可能是并发的。具体到其中的某个节点机, 也完全可以并发地执行分属于不同作业的进程。在这个方面, 整个集群、整个平台作为一个整体的表现与单机上的分时系统是很相似的。所以, 如果某个特定 MapReduce 作业中的 Reduce 节点空闲着, 并不意味着其所在的那台机器是在空转, 因为它也许正在忙于别的什么作业的 map() 计算或 reduce() 计算。

在 Hadoop 的 2.0 版之前, 对于作业的管理是集成的, 有个中央集权的 Jobtracker 在主节点上统管着平台上所有的 MapReduce 作业, 包括作业的部署、启动和进度。这对于规模较小的计算平台是合适的, 但是当平台的规模变得很大的时候就不合适了。所以 2.0 版以后的 Hadoop 采用一种称为 YARN 的机制, 主节点接受一个作业之后就选择一个普通节点, 或曰“从节点”, 在这节点上成立对于这个作业的管理机构, 再由这管理机构操办和管理这个作业的运行, 而主节点则腾出手来专管资源分配。这样就把对于作业的投运和管理也分布出去了。所以主节点的名称以前叫 Jobtracker, 而现在叫 ResourceManager。

除传统的 MapReduce 之外, 新版的 Hadoop 还允许用 Unix/Linux 那些现成的 Utility 程序, 即/bin 等目录下那些工具软件搭建链状数据流, 其每个逻辑上的节点都可以是分布的, 就像 Mapper 一样有很多份, 由很多物理节点并行分担所需的计算。这样就既可以发掘出上下游之间的并行度, 又可以发掘出数据之间的并行度。

1.10 Hadoop 的分布式容错文件系统 HDFS

Hadoop 的文件系统叫 HDFS, 意为 Hadoop 的分布式文件系统。在某种意义上, HDFS

是建立在各节点的宿主文件系统基础上的全局文件系统。如果说 Hadoop 是集群操作系统,那么 HDFS 就是集群的文件系统。

说 HDFS 是分布式文件系统,并不是指整个文件系统中有些文件在这个节点上,有些文件在那个节点上,那样就只是远程 mount 文件卷,没有什么新意了。如果是那样,我们设想有 100 个 Mapper 分布在 100 个节点上,都以同一个文件为输入,只是不同的 Mapper 处理这个文件中不同的数据段(假定每个段的长度是 128MB),于是这 100 个 Mapper 都要到同一个节点上来读数据,哪里还谈得上“数据在哪里,计算就在哪里”?所以 HDFS 的分布方式是沿着文件的长度把它分成许多定长的数据块(每个数据块的长度是 64MB 或 128MB),把不同的数据块存放在不同的节点上。这样,就可以实现“数据在哪里,计算就在哪里”了。当然,如果允许随机写入,那么假如用户要在一个文件中间插入一段数据,就会把原来的数据块边界打乱,那就麻烦了。所以 HDFS 不允许随机写入,而只允许在文件末尾添加(append)内容。对于 OLAP,只允许在文件末尾添加而不允许在文件中间插入或修改是合理的,因为这些数据已是历史数据,是让你拿来分析的,你要修改它干什么?所以,像 YARN 和 MapReduce 框架一样,HDFS 可以说也是专为 OLAP 量身定制的。

光是分布还不行,还得容错。如果没有容错,HDFS 的那种分布方式就是不现实的。设想如果一个文件的内容分布在 100 个节点上,那么只要有其中的一个节点发生故障而变成不可读,这个文件就不完整了。而 100 个节点中有一个发生故障的概率,是单个节点发生故障的概率的 100 倍。也就是说,有可能运行一段时间以后所有的文件都不完整了,这还了得。所以,这种方式的分布必须结合容错措施才有使用价值。

HDFS 的容错措施是把每个数据块都以三个复份分别存储在不同的节点上。这样,如果其中之一发生了故障,就继续使用剩下的两份,并再复制一份,仍维持三个复份。过一会儿,如果那个发生故障的节点又恢复了,那个数据块超过了三个复份,就从中删除一个,总之就是维持在每个数据块三个复份。

对于 HDFS,集群中有一个节点扮演着主节点的角色,称为 namenode。主节点上维持着整个 HDFS 文件系统的目录,可以说是整个 HDFS 的目录服务器,或者说查名服务器。主节点上并不存储数据块复份,而只是存储着所有文件和目录的“元数据”。应用程序需要访问某个文件中某一个位置上的数据时,首先向主节点查询,主节点答以数据块的号码和所存放的节点,并发给一个类似于提货票据的 Token,让应用程序自己去提货。至于主节点的容错,则通过 namenode 的热备来解决。同样的思路也用于数据库 HBase 的设计。

总之,Hadoop 原先是个专门支持 MapReduce 计算模型的大数据处理平台和框架,主要发掘数据间的并行度。现在则也支持链状数据流,这样就既可以发掘出数据之间的并行度,又可以发掘出上下游之间的并行度。从系统结构的角度看,Hadoop 实质上是个建立在宿主操作系统基础上的集群操作系统。经过这些年来的发展,Hadoop 已经得到广泛采用,成为大数据处理平台的“事实标准”之一。所以,我们可以把 Hadoop 作为大数据处理平台的一个标本、一个典型加以研究,就像我们可以把 Linux 作为操作系统的一个标本、一个典型加以研究。把 Hadoop 研究透了,对 Hadoop 有了透彻的理解,再要理解别的大数据处理平台也就不难了。

第 2 章

研究方法

我们的目的是要研究 Hadoop 的源代码,而研究必须有研究方法。这里所说的研究方法是指如何阅读、分析、理解各种计算机程序源代码的方法和手段。实际上对此并没有一种标准的或者公认的方法,各人所用的方法和手段可能都不一样,这里只是把我所用的方法介绍给读者,以期抛砖引玉。

计算机本来就是一个年轻的学科,而如何阅读、分析、理解别人所写的程序,似乎也排不进这个学科,但是却又确实有需要,所以就难免要借鉴一些其他学科的研究方法。在这方面,我认为最值得借鉴的是历史学者们对于文献和史料的研究方法。我们阅读程序代码,就有点像历史学者们阅读历史记载,都是想要从中理出一个脉络、一个来龙去脉,理解当时究竟发生了什么。他们要努力把自己摆进当时的历史环境,力求用当时人的眼光去看待、推测和理解历史上发生的种种事件及其背后的前因后果;我们则要把自己的头脑当成某种语言的虚拟机,例如 Java 虚拟机、C 虚拟机,以理解计算机在执行这些程序的时候究竟会发生什么,以及其背后的逻辑。历史学者们要把各种史料和文献参照引证,从中发现各种事件之间的内在关联;我们也需要参照研究不同程序模块之间的互动,以发现和理解这些模块、这些操作之间的关联。所以,历史学者们常用的研究方法很可能是值得我们借鉴的。

2.1 摘要卡片

在历史学者们的研究方法中,很重要的一个手段就是做摘要卡片,这是历史学者们的一项基本功。这是因为,历史现象庞杂纷纭,如果不针对某个特定的方向进行“去粗存精”的筛选和整理,不抓住重点,就常常会陷入种种琐碎的细节而不能自拔。不过摘要卡片的制作并非一劳永逸,对于同一历史记载,这一次按这个角度、这个粒度所做的卡片,经过一段时间的研究之后可能又要回过去从另一角度、另一粒度再做一个。这,就是历史学者们的调查研究,而结论只能产生于调查研究之后。

当然,时至今日,很少有人会再去抄写那种纸质的卡片了,计算机使摘要卡片的制作更高效、更方便,也更容易检索,但是作为一种研究方法,原理上仍一样,这是值得我们借鉴的。

不过比之历史学者们的研究,我们有个最大的优势,就是一般而言我们可以实验,而他们一般都无法实验。

所以,我们对于程序源代码的研究主要有两种比较有效的手段:一是实验;二是采用摘要的方法浓缩代码以利阅读理解。但是对于像 Hadoop 这样在大规模集群上运行的大型软件,其实想要实验也不容易。即便是在单机上运行的软件,有些实验也并非轻而易举,这也是为什

么测试用例的设计并非小事的原因之一。但是,相比之下,采用摘要结合阅读源代码的研究方法,却是随时随地都可进行的。

其实广义的摘要方法人们早就在用,伪代码就是一种摘要。但是伪代码式的摘要有个缺点,就是常常跟源代码对不上号。因为伪代码和源代码是对同一算法的两种表述,这两种表述有可能相当吻合,也有可能相去甚远。所以还不如直接从源代码中抽取一些在某个角度上具有实质性意义的语句,这样既相当于一种比较真实的伪代码描述,又容易跟源代码对上号,便于互相参照引证。作者平时就是采用这样的方法的,本书的写作也采用了这样的方法。不过从源代码中抽取语句也不能完全原封不动。首先程序中为保持层次清晰而加的 indentation,即逐层后缩,就不宜照搬,因为一般每层右缩一个 Tab,那就是 8 个字符的位置,几个 if 或 for 嵌套下来,一个语句在本行中就排不下了。另外,结合下述的情景分析,这种摘要的一个很大的好处就是可以就地展开函数调用。比方说,函数 A() 里面调用了 B(), 但是 B() 里面又做了些什么呢? 如果我们只能直接引用源代码,那么 A() 和 B() 就得分列,但是实际上 CPU 在执行的时候却是顺序的,先从 A() 里面进入 B(), 做了些什么以后又返回到 A(), 所以此前和此后的程序执行环境就可能有了变化。把 A() 和 B() 分列,一方面对我们理解会有些妨碍,一方面也给叙述带来困难。所以,就地展开函数调用常常是有点好处的,但是当然也不能一概而论。然而既然要就地展开,一方面逐层后缩的问题就更突出了,另一方面还有个问题,就是如何区分因函数调用而形成的层次和因 if、for 等语句所形成的层次,还有因结构成分引起的层次关系,为此最好能用不同的符号表示。

下面用一个实例来说明作者所用的表示方法,这是为 Hadoop 源码中的一个示例程序 WordCount 所做的摘要,侧重于它的 Mapper:

```
class WordCount{
] class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{}
]] Text word    //WordCount 包含 TokenizerMapper,后者又包含 word,所以是双重的包含
]] map(Object key, Text value, Context context)
    > StringTokenizer itr = new StringTokenizer(value.toString())
    > while (itr.hasMoreTokens()) {
    >+ word.set(itr.nextToken())
    >+ context.write(word, one)
    > }
] class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable>{}
]] reduce(Text key, Iterable<IntWritable> values, Context context)
] main(String[] args)
    > Configuration conf = new Configuration()
    > ...    //既然是摘要,自然就会有省略,所以这个省略号可有可无
    Job job = new Job(conf, "word count")
    > job.setMapperClass(TokenizerMapper.class) //想知道这是怎么回事,所以就地展开
        == Job.setMapperClass(Class<?extends Mapper> cls)
    >> ensureState(JobState.DEFINE)//job 的状态必须还在 DEFINE 阶段,否则发起异常
    >> conf.setClass(MAP_CLASS_ATTR, cls, Mapper.class)
```

```

== Configuration.setClass(String name, Class<?> theClass, Class<?> xface)
//把 cls 即 TokenizerMapper.class 写入配置块中以 MAP_CLASS_ATTR 为键的配置项
> job.setReducerClass(IntSumReducer.class)
> System.exit(job.waitForCompletion(true)?0 : 1)

```

这是对于 WordCount 这个类的定义,我用一对花括号表示类,或界面(interface),或枚举(Enum),就像用圆括号表示函数(或方法,在本书中函数与方法是同义词,可以互换使用)、用方括号表示数组(Array)。下面会讲到,类就相当于数据结构加函数,再加对于内嵌类的定义。所以类的内部可以有数据成分、有函数定义、有对别的类的定义,我用右方括号来表示这种包含关系,因为这是键盘上最接近于“包含”这个语义的符号。包含的关系可以嵌套。同时,我用“>”表示调用或执行的关系,也以此表示因调用或执行而形成的层次关系。这样,通常一个 Tab 制表位或至少两个空格的右缩就减少成一个“>”。此外,我还用一个加号表示因 if、for、while 等程序结构所形成的层次关系,读者可以看到这里 while 循环中的语句前面都多了一个加号。其实使用加号并不理想,因为在遇上例如“++n”这样的语句时容易让人看花了眼,但是键盘上也没有明显比这更合适的符号。

WordCount 这个类没有数据成分,或者这里被省略了,但是有对于 TokenizerMapper 和 IntSumReducer 两个内嵌类的定义,还有个 main()函数。作为一个类,一般应该有一个构造方法,也叫 WordCount(),但是在摘要中往往被省略,因为构造方法的代码通常都很简单。但是 WordCount 这个类的源码中倒确实没有构造方法,这应该是因为它没有数据成分,无须构造。

但是 TokenizerMapper 内部确实有个数据成分 word,其类型为 Text,还有个 map()函数。TokenizerMapper 也没有提供构造方法,应该是因为它的数据成分 word 无须初始化。这个类是对 Mapper 类的扩充,而 Mapper 类的定义是一种“模板(Template)”定义,又称“泛型定义”,对此缺乏了解的读者可以先看一点 Java 语言编程方面的参考书。

现在,假定我们想知道 main()函数中对 job.setMapperClass()的调用是怎么回事,首先我们得知道对象 job 的类型是 Job,所以这是对 Job.setMapperClass()的调用,于是我用“==”表示这样的等价关系。另外,比较好的做法是把定义于这个函数的形式参数表抄列在这里,让实参和形参形成对照。但是说实话我也常常偷懒,因为有时候这种对应其实很明显。然后,这里就把这个函数就地展开了,我们可以看到这个函数中的代码摘要。如果需要,我们还可进一步展开 Configuration.setClass()。

至于摘要中的黑体加粗部分,则纯粹就是为了引起注意,没有别的含义。

摘要是很个人、随机性很大的工作,同一个人对同一段代码所做的两次摘要,哪怕是为了同一个目的、从同一角度,也可能会有所不同,就像历史学者所做的摘要卡片一样。因为摘要 是给人看(而不是让机器处理)的,而且主要是给自己看的,只要自己认为合适就好。当然也要尽量接近程序的原意。

我们在做摘要时原则上要尽量保持语句的原貌,但是有时候也不得不对代码进行某种等价变换。这一般发生在几种情况下。一种是函数调用的嵌套,例如上面摘要中的语句:

```
System.exit(job.waitForCompletion(true)?0 : 1)
```

这是根据 job.waitForCompletion(true)的返回值确定以常数 0 或 1 作为 System.exit()的

参数。如果我们想展开 `job.waitForCompletion()`，就地放在这个语句下面就不合适了，因为日后可能一下子搞不清这究竟是对 `job.waitForCompletion()` 还是对 `System.exit()` 的展开。在这样的情况下，我们就得将其变换成例如下面这样：

```
w = job.waitForCompletion(true)
System.exit(w?0 : 1)
```

这样，就可以就地展开了。这里的中间变量名 `w` 是任意的，当然如果能用例如 `success` 之类有点意义的就更好。

还有一种情况是函数调用出现在 `if`、`while` 等语句的条件部分，就如上面 `map()` 函数中的 `while (itr.hasMoreTokens())` 语句。碰到这样的情况，如果真的想要就地展开，就得把代码处理成例如这样：

```
> h = itr.hasMoreTokens()
> while (h) {
>+ ...
>+ h = itr.hasMoreTokens()
> }
```

或者，就不要就地展开了，改成单独为此做一代码摘要加以展开，例如：

```
[WordCount.TokenizerMapper.map() > hasMoreTokens()]
```

```
StringTokenizer.hasMoreTokens()
> ...
```

这里，前面方括号里面是调用路径，说明这是从 `map()` 函数中调用过来的。调用路径中的“>”表示直接调用。有时候我们也需要说明间接或辗转的调用关系，或者是“引起、导致”的关系而不是调用的关系，那就用“=>”。

下面则是对 `hasMoreTokens()` 这个函数的代码摘要，里面对于函数调用还可以再就地展开。

还有一种比较特殊的情况，那是对某些抽象类的动态扩充或对界面(interface)的动态实现，这其实并不是我们做摘要时的就地展开，但是形态上有点相似并对摘要的制作有点影响。我们举个实例。先看原始的代码(与当前话题无关的内容已经省略)：

```
class StreamPumper {
...
StreamPumper(final Log log, final String logPrefix,
              final InputStream stream, final StreamType type) {
...
thread = new Thread(new Runnable() {
@Override
public void run() {
try {
```



```

        pump();
    } catch (Throwable t) {
        ShellCommandFencer.LOG.warn(
            logPrefix + ": Unable to pump output from " + type, t);
    }
}
}, logPrefix + ": StreamPumper for " + type);
thread.setDaemon(true);
}
...
}

```

这里,在 StreamPumper 类的构造方法 StreamPumper()中,要创建一个线程来执行一个 Runnable 对象,但是 Java 中的 Runnable 是个界面(interface),对于界面是无法创建对象的,必须要定义一个实现这个界面的类,才可以创建那个类的对象。所以一般的做法是老老实实定义一个类来实现这个界面,例如“class StreamPumperImpl implements Runnable{ }”,这叫静态定义。但是有时候觉得这太麻烦了,就在程序中动态定义一个无名类,并立即加以创建。所谓动态定义,就是要补上这个界面要求提供的方法,或者用自己的方法替换继承来的方法。所以这里就提供了一个 run()函数,里面调用 pump()。

为这么一段代码做的摘要,可以是这样:

```

class StreamPumper {
] StreamPumper(Log log, String logPrefix, InputStream stream, StreamType type)
    > hread = new Thread(new Runnable() {
        ] run()
        > pump()
    > thread.setDaemon(true)
}

```

我们原则上把 run()函数放在“new Runnable()”的下方,表示这是所动态定义的那个实现了 Runnable 界面的无名类的 run()函数。不过在这样的情况下我们一般就不对 pump()进行就地展开了。

顺便提一下,在构造函数 StreamPumper()的参数表中,我们把 final 一类的关键字省略了,同样我们一般也省略 public、private 一类的关键字,因为那些属性无关程序的逻辑和流程。更重要的是,除非实有必要,我们也会省略 try{} catch()的程序结构,因为对于异常的处理一般不属于程序的“主旋律”。另外,一些对于边界条件的检验(常常称为 sanity check),例如“if(n==0) return”之类,也往往与程序的主旋律无关。还有一些善后操作,例如“close(file)”等,对于我们理解程序的流程和原理没有多大关系,所以也常被省略。

本书很少直接引用原始代码,而大多采用摘要。这样一来紧凑,二来可以就地展开。还有一个好处,就是可以降低对于版本升级的敏感度,因为版本升级反映在具体的函数上有时候只是某些细部的修改(特别是对于 Bug 的修理)而无关宏观的流程,不一定在摘要里面表现出来。

2.2 情景分析

正如对于历史事件的研究一定要把它放在当时的整个时代背景中并力图理解它的前因后果一样,对一段具体程序代码的阅读理解也要把它放在更为宏观的背景和过程中,把它看成某个宏观过程中的一个片段或者一个侧面,这样才能搞清它的来龙去脉。这样的研究方法和叙述方式就是情景分析。我们知道学外语有“情景会话”的方法,就是不要一味地背单词、学语法,而要以真实或假想的情景把所学的单词和语法串起来,这里面既有情节又有背景和场景,这样学效果要好得多。对程序代码的情景分析也是一样的道理。

我之所以用“情景”这个词而不是“场景”,是为了突出其动态的、过程的、情节的一面,而避免给人以“瞬间的场面和景象”这样的理解。对于一个事件,我们拍个照片,就可以说这是当时的一个场景;但是,如果要说是情景,那就非得拍一段视频不可了。

所以,我们对程序代码的分析和研究,应该尽量结合具体的情景,作为过程来加以考察。其实,“程序”这个词本来就有“过程”的意思。

Hadoop 的代码是用 Java 语言编写的,Java 是一种面向对象的程序设计语言,Hadoop 的代码就是按照面向对象的风格和方法编写的。根据作者的体验,情景分析对于面向对象的程序代码的分析理解尤显重要。这是为什么呢?所谓“面向对象(Object Oriented,OO)”其实就是面向个体,或者面向“类(class)”。在这种方法中,每个类,以及类中的每个 Object,即个体,都是独立自治的,只要设计好每个类的个体如何与外界互动,拟出每个类的“规格书”就行了。具体某个类的开发者则只要按规格书编写程序并测试就行,无须也不太可能了解整个软件产品在实际运行中的流程。这对于提高软件开发的劳动生产率、提高软件质量、加强对开发过程的管理,无疑有着十分重要的意义。这也正是大工业生产的规律。这就好比,汽车制造厂里管刹车系统的人不必了解引擎是怎么工作的、方向盘是怎么工作的,只要按规格书做好刹车系统就行了。产品虽然不同,道理是一样的,面向对象的程序设计就特别适合大型软件的分头开发,特别适合把大型的系统分解成许多部件,然后把任务分配给许多不同的小组加以开发和实现。然而,对于系统的研究者,对于努力想要理解整个软件产品的工作原理的人,这样却既有好的一面也有不利的一面,可能不利的一面还更突出一些。这不利的一面,就来自缺少一条主线把这许许多多的类给串起来。

打个比方,面对 Hadoop 这一大堆的类,我们就像拿着一本药典,里面有关于每种药物的描述,却不知道对于具体的疾病该怎样从中选择用药和配方,这时候我们更想看到的是具体患者的病情以及医生如何诊断和用药。再打个比方,如果《水浒》中只有对于数百个人物的类似于说明书和规格书一样的描述,例如“武松,身高几许,体重几许,臂力达到多少公斤,正义感强,性急躁,等等,等等”,而没有故事没有情节,那我们把整本书看上几遍也不知道发生了什么。

所以,对于面向对象的程序代码,情景分析的方法就更显得重要。

2.3 面向对象的程序设计

如上所述,Hadoop 的代码是用 Java 语言编写的,而 Java 是一种面向对象的程序设计语

言, Hadoop 的设计和实现也确实是对面向对象的。那么, 所谓面向对象, 究竟是什么意思? 这个概念是怎么来的呢? 这个问题, 许多天天在用 Java 编程的人也未必清楚, 但是对于我们分析和理解 Hadoop 的代码却是有意義的, 因而值得用一些篇幅做点介绍。

最早的程序是用汇编语言甚至机器语言编写的, 程序员可以使用 CPU 指令系统中的任何指令, 那样编程效率既很低, 又不安全, 程序的可读性就更不用说了。这就迫使人们发展出高级语言, 以提高编程效率。但是那时候的程序还都是所谓“spaghetti code”, 或者说是“炒面式代码”, 乱成一团。

后来荷兰科学家 Dijkstra 发表了著名的论文“Go To Statement Considered Harmful”, 主张在高级语言中去除 goto 语句。结合着“去 goto”, 人们又提出“结构化程序设计”的概念和方法, 论证只要有 if-then-else 和 while 这些程序结构, 就完全可以不要 goto。

事实上此后的许多高级语言中确实是不提供 goto 语句的。但是请注意, 这只是不让在采用高级语言的应用程序中使用 goto 语句, 而并不限制在采用例如 C 语言和汇编语言的系统程序中以及经编译产生的可执行代码中使用。其实这就是把使用 goto 语句或指令的人群范围缩小了, 只让比较专业的人员使用, 而限制公众使用。这样的情况也发生在别的场合。例如, 大家知道 Java 语言不允许使用指针, 但是就有人(当年作者就是其中之一)感到困惑了: 不使用指针, 那好多功能就无法实现啊? 再说, Java 中的所谓 Reference 与指针究竟有什么区别呢? 其实 Java 只是不让 Java 程序员使用指针, 它自己还是用的, Java 程序经编译后的代码中当然也用指针, 要不然它那些功能怎么实现? 它只是不让你们 Java 程序员用指针而已, 这是为你们好。至于 Reference, 也确实就是指针, 只是赋值时它要检查是否真的指向一个所述类型的对象。应该承认, 这确实比 C 语言安全多了, 从而程序员的工作效率也可提高, 并且还降低了程序员入门的门槛。

与此相平行, Pascal 语言之父 Nicklaus Wirth 提出了“算法 + 数据结构 = 程序 (Algorithms + Data Structures = Programs)”的命题, 提醒人们不要光把眼光盯在算法上, 数据结构同样重要。事实上, 程序中的许多 bug 都来自对数据结构的不当访问。这就好比, 对于一个设备, 哪怕是很简单的物件, 要是谁都可以来自助使用, 七手八脚, 就难免出错。比较好的办法是凡要使用就必须按规定的方法使用, 不能让人自作主张。更好的办法是有专人负责, 你要怎么样就对他说, 由他来帮你操作。

在这样的背景下, Hoare 和 Brinch Hansen 等人提出和发展了“模块化程序设计”、“结构化程序设计”的概念和方法, 提出了对象即 Object 的概念, 说 Object 就是“数据结构, 和定义于这个数据结构上的全部操作”。而“类(class)”, 则是对具有相同数据结构和相同操作方法集合的那些 Object 的抽象。

所以, 当我们说创建了一个某类的对象时, 我们说的是: 分配存储空间用作这个类所定义的数据结构, 并带有这个类所定义的对于这个数据结构的全部操作(有些操作也可以是超出这个范围的)。这样, 我们就把数据“封装”在具体的对象中了。但是这又要靠语言来保证, 所谓靠语言保证, 其实是靠这个语言的编译器(或解释器)保证。例如, 在编译的过程中, 如果发现某个类的程序中要自行直接访问另一个类的某个数据成分, 而那个数据成分的性质在程序中说明为 private, 就马上报出错并拒绝继续编译, 这样就使该项数据的封装得到了保证。读者也许会问, 要是我绕过编译, 直接用 Java 的机器码写个程序来访问这个数据成分呢? 答案是: 如果你有这个本事, 所在的公司或团队又允许你这样做, 那当然是挡不住的, 但是又有几个人

会这样做呢?

不过光是做到了数据封装还不足以让一个语言成为“面向对象”,人们普遍认为需要满足三个要求才算是面向对象,还有两个是“多态(polymorphism)”和“继承(inheritance)”。如果不同时满足这三项要求,就只能说是“基于对象的(Object Based)”,而不能说是“面向对象的(Object Oriented)”。

不过也有的类中并未定义数据成分,而只是定义了一些方法,这样的类实际上就是一个小小的库函数(lib)模块。反过来,如果只定义了数据成分而并未定义任何方法,那就只是一个纯粹的数据结构,但是这样的情况几乎没有。

所谓“多态”,有几方面的意思。首先是同一个函数名可以用于多个不同的函数,只要参数的类型或个数不同,比方说,同样是在 BlockManager 这个类里面,就定义了两个都叫 completeBlock 的函数:

```
completeBlock(final BlockCollection bc, final int blkIndex, boolean force)
completeBlock(final BlockCollection bc, final BlockInfo block, boolean force)
```

二者的区别仅在于第二个参数的类型,一个是 int,另一个是 BlockInfo。之所以可以如此,是因为 Java 的编译器会自动把函数的参数表,即所有各个参数的类型,加以排列编码作为后缀拼接在函数名后面。这样,程序员所看到的函数名与编译器所使用的函数名其实是不一样的。这对于程序员确实是提供了方便,因为这二者的功能是一样的,符合思考的习惯。用 C 语言编程的时候,像这样的情况就得用不同的函数名,例如 completeBlock_Index() 和 completeBlock_Info(),那就要麻烦一些,所以这确实是一项改进。可是这对于程序的阅读分析却未必是好事,因为阅读分析者看到某处调用了 completeBlock(),却不知道究竟是其中的哪一个,这时候就得仔细去看调用时的实参分别是什么类型,再去与两个函数的参数表比对,才能知道究竟是调用了哪一个。所以,在阅读分析面向对象的程序代码时,很容易在这样的情况下误入歧途,需要特别注意。

“多态”的另一种意思是,一个类可以有多个不同的“子类(Subclass)”,在程序中可以用“父类”来泛指不同的子类(也许说“大类”、“小类”更符合我们的习惯)。比方说,Hadoop 的代码中定义了一个抽象类 InputFormat。所谓抽象类,是指这个类声称要提供的函数有些是尚未实现的、是悬空的,有待于它的子类加以落实,因为预知不同的子类应该会有不同的实现;而 InputFormat 的子类,即扩充了 InputFormat 的类,事实上就有十多个。然后,程序中可以对 InputFormat 类进行某些操作,而无须明说这是对于哪个具体的子类。其实这样的“动态绑定”在 C 程序中也有使用,例如 Linux 内核中的 file_operations 数据结构就是用于这种目的,但是在 C 程序中一般用得不多,而在 Hadoop 这样的大型 Java 程序中就比比皆是了。和上述函数名的多态一样,我们在阅读分析 Java 程序时也很容易因此而误入歧途。

要成为面向对象的程序设计语言,还须满足“继承”的要求。这是说,如果一个类声称扩充(extends)了另一个类,那么后者就是它的父类,于是它就自动继承了父类的所有成分和方法,父类中有什么它就有,然后它还可以再补充定义一些成分和方法。这样,当你看着一个类的代码时,一定要意识到这也许不是它的全部,它可能还从父类那里继承了不少内容。另外,当你看到程序中某处调用了这个类的某个方法,可是它的定义中没有这么一个方法的时候,一定要记得去它的父类甚至祖类那里去寻找。

所以,作者的体验是,比之 C 程序,我们阅读分析 Java 程序的时候要多长个心眼。

2.4 怎样阅读分析 Hadoop 的代码

如前所述,阅读分析 Hadoop 的代码,无非就是两种手段:一是实验;二是阅读,包括做摘要。所谓实验,主要是通过设断点等调试手段跟踪程序的执行,也包括设计测试用例让系统定向产生计算结果和运行日志以供分析。但是,对于像 Hadoop 这样的大型软件,要实验也不容易,实际上也无法覆盖太多的用例,所以还是要结合阅读分析。而阅读分析,则最低限度的要求是有个能做关键字搜索的工具。对此最原始的当然是 grep,但是对 Hadoop 使用 grep 已经让人感到不太方便,最好能用像 Eclipse 或 Visual Studio 一类的 IDE 即综合开发环境。能在上面重新编译 Hadoop 并且运行那是最好,要不然就把它当成搜索工具也行。其实纯粹搜索也不坏,因为搜索的覆盖面是最大的,也无须事先把 Hadoop 编译一遍,当然缺点是有用信息容易被淹没,效率也比较低。

还要注意一点,采用 Maven 作为编译和构建工具之后,新版 Hadoop 的目录嵌套极深,以至于搜索时的路径可能因超长而被截尾,使搜索的范围不全而造成误导。

最后,说到底,阅读理解是人的事,机器和工具只能使你提高效率,最终的理解还是靠人。

作者假定本书的读者学过 Java 语言。但是也尽量照顾一些用过 C/C++ 但刚开始在学习和熟悉 Java 的读者,这些读者不妨从第 5 章即作业提交那一章入手,以后对 Java 多熟悉一点再回头看第 3 章和第 4 章。另外,我在第 5 章中引用源代码会多一些,以后就逐渐转入以摘要为主。

第 3 章

Hadoop 集群和 YARN

3.1 Hadoop 集群

虽然 Hadoop 也可以在单机上运行,但是这个平台的典型运行场景无疑是在多机的集群(Cluster)上。我们把运行着 Hadoop 平台的集群,就 Hadoop 平台的边界所及,称为“Hadoop 集群”。其中的每台机器都成为集群的一个“节点(node)”,节点之间连成一个局域网。这个局域网一般都是交换网,而不是路由网。这就是说,集群中只有交换机(switch),一般是二层交换机,也可能是三层交换机,但是没有普通的路由器,因为那些路由器引入的延迟太大了。不过这也不绝对,有时候可能确实需要将一个集群分处在不同网段中,而通过路由器相连,但是这并不影响 Hadoop 的运行(除性能降低之外)。就 Hadoop 而言,路由器与交换机在逻辑上是一样的。

集群内的节点之间可以通过 IP 地址通信,也可以通过节点的域名即 URL 通信,这就需要有 DNS 的帮助。这意味着,在网络可以通达的某处存在着 DNS 服务,因而可以根据对方的 URL 查到其 IP 地址。这也意味着,集群内这些节点都应有个域名,并且登记在 DNS 中。实际上节点间还可以通过节点名通信,但是那跟 URL 本质上是一样的,最后总是转换成 IP 地址。不管是静态设定还是通过 DHCP 动态分配,每个节点都必须有个 IP 地址。

既然可能需要通过域名互相访问,这些节点就得与 DNS 打交道。一般涉及 DNS 的操作都在操作系统或底层的库程序中,对于应用层是透明的。比方说我们通过 HTTP 访问网站就只需要提供其域名,而 HTTP 驱动层自然会与 DNS 服务器交互以获得目标网站的 IP 地址。但是 Hadoop 不能甘于 DNS 对其保持透明,因为它的有些操作需要知道具体节点的 IP 地址、域名之类的信息,因此需要直接与 DNS 交互,为此 Hadoop 定义了一个名为 DNS 的类,用来提供帮助。

```
class DNS{}
```

```
    String cachedHostname = resolveLocalHostname()
```

```
    String cachedHostAddress = resolveLocalHostIPAddress()
```

```
    String LOCALHOST = "localhost"
```

```
    reverseDns(InetAddress hostIp, String ns) //ns: The host name of a reachable DNS server
```

```
    //这是逆向查询,从 IP 地址查其域名。参数 ns 为 DNS 服务器的地址
```

```
    > String[] parts = hostIp.getHostAddress().split("\\.")
```

```
    > String reverseIP = parts[3] + "." + parts[2] + "." + parts[1] + "." + parts[0] + ".in-addr.arpa"
```

```

    //倒排 IP 地址的 4 个字段,例如“110. 22. 33. 4”就变成“4. 33. 22. 110. in-addr.arpa”
> DirContext ictx = new InitialDirContext()
> attribute = ictx.getAttributes(
    "dns://" + ((ns == null) ? "" : ns) + "/" + reverseIP, new String[] { "PTR" })
    //形成一个 DNS 查询语句
> String hostname = attribute.get("PTR").get().toString()
> hostnameLength = hostname.length()
> if (hostname.charAt(hostnameLength - 1) == '.') {
>+ hostname = hostname.substring(0, hostnameLength - 1)
> }
> return hostname
] getIPs(String strInterface, boolean returnSubinterfaces)
    //获取绑定在某个网口(例如 eth0 或 eth0 : 0)上的所有 IP 地址
] getDefaultIP(String strInterface) //获取绑定在某个网口上的默认 IP 地址
] resolveLocalHostname() //获取本机的主机名
> String localhost = InetAddress.getLocalHost().getCanonicalHostName()
> return localhost
] resolveLocalHostIPAddress() //获取本机的 IP 地址
> String address = InetAddress.getLocalHost().getHostAddress()
] getDefaultHost(String strInterface, String nameserver) //参数 nameserver 为 DNS 服务器地址
> if ("default".equals(strInterface)) return cachedHostname
> if ("default".equals(nameserver)) return getDefaultHost(strInterface)
>> return getDefaultHost(strInterface, null) //将第二个参数设成 null
> String[] hosts = getHosts(strInterface, nameserver)
> return hosts[0]
] getHosts(String strInterface, String nameserver) //获取绑定于网口 strInterface 的所有域名
    //参数 nameserver 为 DNS 服务器地址或 IP 地址,可以是 null
> String[] ips = getIPs(strInterface) //获取该网口的所有 IP 地址
> Vector<String> hosts = new Vector<String>()
> for (int ctr = 0; ctr < ips.length; ctr++) { //逐一查询绑定于这些地址的域名
>+ hosts.add(reverseDns(InetAddress.getByName(ips[ctr]), nameserver))
> }
> return hosts.toArray(new String[hosts.size()]) //返回这些域名

```

这个类没有构造函数,实际上也未创建对象,而只是作为类似于程序库那样的模块存在。模块外部需要调用其某个函数时,就通过例如 DNS.getDefaultHost() 这样的方式加以调用。这个模块也有几个数据成分,用来缓存本机的主机名和 IP 地址。

在 Hadoop 涉及 DNS 的代码中,这些就已经是最底层的了,再往下就由 Java 语言的 SDK,即 JDK 提供了。例如 InetAddress.getLocalHost(),这里的 InetAddress 就是从 JDK 导入的,在源码文件 DNS.java 的前面有“import java.net.InetAddress”。

举个使用的实例。在 HDFS 子系统内 DataNode 的代码中,有个方法函数 getHostName(),

用来获取本节点的主机名,其实现是这样的:

```
private static String getHostName(Configuration config) throws UnknownHostException {
    String name = config.get(DFS_DATANODE_HOST_NAME_KEY);

    //先看看在配置文件中是否已有设定
    if (name == null) {
        //如果没有设定,就求助于 DNS 服务
        name = DNS.getDefaultHost(
            config.get(DFS_DATANODE_DNS_INTERFACE_KEY,
                DFS_DATANODE_DNS_INTERFACE_DEFAULT),
            config.get(DFS_DATANODE_DNS_NAMESERVER_KEY,
                DFS_DATANODE_DNS_NAMESERVER_DEFAULT));
    }
    return name;
}
```

Hadoop 允许在配置文件中设定本节点的主机名,所以先要看看 Configuration 中是否有这样的配置项,如果没有就求助于 DNS 服务。整个过程涉及 5 个字符串常数:

```
String DFS_DATANODE_HOST_NAME_KEY = "dfs.datanode.hostname"
String DFS_DATANODE_DNS_INTERFACE_KEY = "dfs.datanode.dns.interface"
String DFS_DATANODE_DNS_INTERFACE_DEFAULT = "default"
String DFS_DATANODE_DNS_NAMESERVER_KEY = "dfs.datanode.dns.nameserver"
String DFS_DATANODE_DNS_NAMESERVER_DEFAULT = "default"
```

可见,如果配置文件中设定了本节点主机名,则配置项名称是“dfs.datanode.hostname”。然而我们在配置文件中搜索不到这个配置项,所以并未设定。下面的配置项“dfs.datanode.dns.interface”倒是有,其值为“default”。这个配置项本应是网卡名称,现在采用默认。下一个配置项是 DNS 服务器,我们在前面看到 DNS.getDefaultHost()的第二个参数是 DNS 服务器的主机名 nameserver;这也是可以通过配置文件设定的,配置项名称为“dfs.datanode.dns.nameserver”。事实上,hdfs-default.xml 中有这个配置项:

```
<property>
  <name>dfs.datanode.dns.interface</name>
  <value>default</value>
  <description>The name of the Network Interface from which a data node should
    report its IP address.
  </description>
</property>
```

配置项的值为 default,因而 DNS.getDefaultHost()会将其替换成 null。没有 DNS 主机名怎么办呢?那也不要紧,操作系统或 DNS 驱动自会将其补上。

这里应该说明,代码中的 config 是个 Configuration 类的对象,我们不妨称之为“配置块”。配置块中的信息主要来自配置文件,但也可以在程序中动态加以设置或改变。所以,配置信息主要来自配置文件,但不一定完全来自配置文件。

以上所述都不涉及集群的构成和拓扑。说到 Hadoop 集群的构成和拓扑,这就有点复杂了。Hadoop 的代码中定义了几个相关的 class,用来反映和管理集群的拓扑。Hadoop 之所以需要有集群的构成和拓扑信息,是因为这些信息对于合理安排数据存储和计算、对于容错、对于提高效率、都是颇为重要的。

在实际投入运行的集群中,节点机一般不太会是普通的台式 PC 机,而会是装在机架(rack)上的服务器,尤其是所谓“刀片式(blade)”服务器。不过我们并不关心其是否刀片式,关心的是:一般同一个机架上的服务器总是连在同一个交换机上,所以同一机架上的两个节点之间通信只需流经一台交换机,但是不同机架上的节点间通信就可能要流经至少两台交换机了。另外,容错的关键在于冗余,假定我们要把一个节点上的某些数据冗余存储在另一个节点上作为备份,那就不应该选择同一机架上的节点,要不然两个节点说不定就同时断电了。

现在的交换机都是可网管的,有自己的 IP 地址,所以 Hadoop 把交换机也看成网络中的节点。而机架也可能是可网管的,或者就让交换机同时代表着机架,因为机架上往往集成着交换机。

这样,如果把交换机和机架也看成节点,那么整个局域网或者整个集群的拓扑就是个树状的结构。Hadoop 集群中能实际参与数据处理的节点都是这棵树上的叶节点,树的根节点就是整个局域网的入口,通常这是一台路由器。其余的节点则都是“中间节点”或“内部节点”,那都是交换机或机架。

Hadoop 需要了解这些节点的基本情况以及连接的拓扑,所以定义了几个 class 来反映实际的情况。首先是反映节点基本情况的 NodeBase,下面是其定义的摘要:

```
class NodeBase implements Node {}  
] static char PATH_SEPARATOR = '/'  
] static String ROOT = ""  
] String name; //host : port#  
] String location; //string representation of this node's location  
] int level; //which level of the tree the node resides  
] Node parent; //its parent  
] NodeBase(String name, String location, Node parent, int level)  
] getPath(Node node)  
  
> return node.getNetworkLocation() + PATH_SEPARATOR_STR + node.getName()
```

请读者记住,既然是摘要,就肯定不完整,只是把其中我们可能感兴趣的内容摘出来了。例如构造函数 NodeBase(),根据调用参数表的不同就有好几个,这里只列出了其中信息最详尽、参数最多的一个。

我们先看其数据部分:有 name 即节点名;有 location 即其所在的位置或路径,也就是从根节点往下以“/”分隔的节点路径,类似于文件路径,例如“/east/row8/rack3”就是东区 8 排 3 号机架;还有 level 即节点所处的层次;还有该节点的父节点 parent,那应该是机架或交换机。注意,parent 的类型是 Node,但实际上 Node 是界面而不是类,这表示任何实现了 Node 界面的类都是可以的,而 NodeBase 就实现了这个界面。

再看其方法部分,这个类实现了 Node 这个界面(interface),所以至少会提供 Node 界面所规定的所有方法函数。不过这里只列出了一个 getPath(Node node)。给定一个实现了 Node

界面的某类对象, `Nodebase.getPath()` 返回其在局域网中的全路径, 比方说可能是 “/east/row8/rack3/Node_A”, 如果节点名是 “Node_A” 的话。

可想而知, 集群所在的局域网中的每个节点都会有个 `NodeBase` 对象。下一步就是它们互连的拓扑了。为此 Hadoop 定义了一个类叫 `NetworkTopology`:

```
class NetworkTopology {}
[ static String DEFAULT_RACK = "/default-rack" //默认的机架节点名
[ static int DEFAULT_HOST_LEVEL = 2 //树的高度默认为两层
[ InnerNode clusterMap //这是整个集群(局域网)的拓扑图
[ int numOfRacks //集群中共有几个机架
[ class InnerNode extends NodeBase {}
[ List<Node> children = new ArrayList<Node>() //递归构成整个集群的拓扑
[ int numOfLeaves //本子树有几个叶节点
[ isRack()
[ isAncestor(Node n)
[ isParent(Node n)
[ add(Node n) //Add node n to the subtree of this node,
//这是 NetworkTopology.InnerNode.add()
[ add(Node node) //Add a leaf node,这是 NetworkTopology.add()
[ remove(Node node)
[ contains(Node node) //集群中是否含有这个节点
[ getDatanodesInRack(String loc) //获取指定机架上的所有节点
[ getNode(String loc) //获取这个节点对象
[ getNumOfRacks() //集群中共有几个机架
[ getNumOfLeaves() //有几个叶节点
[ getDistance(Node node1, Node node2) //计算任意两个节点之间的拓扑距离
[ isOnSameRack( Node node1, Node node2) //两个节点是否在同一机架上
[ getWeight(Node reader, Node node) //节点 node 离节点 reader 的远近
// 0 is local, 1 is same rack, 2 is off rack
```

`NetworkTopology` 的数据成分主要就是 `clusterMap`, 这是一个 `InnerNode` 类对象, 而 `InnerNode` 类就定义于 `NetworkTopology` 内部。逻辑上 `NetworkTopology.clusterMap` 相当于根节点的 `InnerNode` 对象, 实际上根节点是虚的, 因为集群内并没有这么一个节点, 那就可能是一条网线。但是 `clusterMap.children` 是个 `List`, 其中的元素可以是 `InnerNode`, 也可以是 `NodeBase`。若是 `InnerNode` 则还有自己的 `children`, 这样就递归构成了整个集群的拓扑。

然而拓扑中的信息是怎么来的呢? 怎么知道哪个交换机或机架上有些什么节点? 这主要靠各个节点主动向相关子系统的主节点登记, 最终还是来自各个节点的路径配置。

Hadoop 的代码中还定义了另一个类, `NetworkTopologyWithNodeGroup`, 这是对 `NetworkTopology` 的扩充, 下面仅摘取二者相异的部分:

```
class NetworkTopologyWithNodeGroup extends NetworkTopology {}
[ ...
```

```

] isNodeGroupAware()
    > return true
] isOnSameNodeGroup(Node node1, Node node2)
] getWeight(Node reader, Node node)
    //0 is local, 1 is same node group, 2 is same rack, 3 is off rack
] class InnerNodeWithNodeGroup extends InnerNode {}
]] isNodeGroup()

```

代码中有注释,说明这是针对 4 层结构的,而 NetworkTopology 则是针对 3 层结构的。比较二者的函数 getWeight() 的返回值,对于 NetworkTopology,返回值是 0 至 2,而对于 NetworkTopologyWithNodeGroup 则是 0 至 3。这多出来的一层应该是在机架内部,把节点又分成了组。

那么究竟是 3 层还是 4 层呢? 这要由系统管理员根据具体的集群网络来加以配置。Hadoop 的代码中包含了一些用于系统配置的.xml 文件(以 XML 语言编写)。其中有个文件 core-default.xml,在目录 common-project/hadoop-common/src/main/resources 中,就是一个十分重要的配置文件。当然,同样也很重要的配置文件还有好多。

配置文件中有很多配置项,每个配置项都是一个 KV 对,即“键/值对”,均以字符串模式存在于配置文件中。Hadoop 的程序要用到具体配置项时就从配置块中读取,而配置块的信息则来自配置文件。以 NetworkTopology 对 NetworkTopologyWithNodeGroup 为例,我们可以了解 Hadoop 是怎样使用这些配置项的。当 Hadoop 需要创建一个网络拓扑对象,但不确定该是 NetworkTopology 还是 NetworkTopologyWithNodeGroup 时,程序中调用的是 NetworkTopology.getInstance(),虽然这是 NetworkTopology 的 getInstance(),但实际上并不局限于此,从而提供了灵活性:

```

[NetworkTopology.getInstance()]

public static NetworkTopology getInstance(Configuration conf){
    return ReflectionUtils.newInstance(
        conf.getClass(CommonConfigurationKeysPublic.NET_TOPOLOGY_IMPL_KEY,
            NetworkTopology.class, NetworkTopology.class), conf);
}

```

这个函数返回的可以是 NetworkTopology 类对象,但也可以是继承、扩充了 NetworkTopology 的某个子类的对象,如 NetworkTopologyWithNodeGroup。具体的对象是通过 ReflectionUtils.newInstance() 创建的,而究竟是创建哪一个类的对象则临时由 conf.getClass(),即 Configuration.getClass() 确定。这个函数的第一个参数为配置项名称,它会在配置块中寻找该配置项的值,这就确定了实际要创建的是哪一个类的对象。如果配置块中没有这样的配置项,那就以第二个参数所给定的类作为默认,那就是 NetworkTopology.class。注意,NetworkTopology.class 并非一个 NetworkTopology 类对象,而是一个 Class 类对象,这是对于 NetworkTopology 这个类的描述。如果找到了配置项的值,就是目标类的路径名,那还得核对一下第三个参数,在这里也是 NetworkTopology.class。这个参数本应是对某个界面

(Interface)的描述,也可以是对某个基础类的描述,看所给定的类是否实现了这个界面,或者是否是对该基础类的扩充。

这里我们就看到 Java 语言的 reflect 机制的好处(之一)了。如果没有 reflect 机制,这里一般而言就会需要好多类似于下面这样的 if 语句:

```
if(...) ret = new NetworkTopology(...);  
else if (...) ret = new NetworkTopologyWithNodeGroup(...)  
else if (...) ...
```

那么这里的配置项名称是什么呢?在 CommonConfigurationKeysPublic 这个类中定义了许多常数字符串,其中之一就是 NET_TOPOLOGY_IMPL_KEY:

```
String NET_TOPOLOGY_IMPL_KEY = "net.topology.impl"
```

这个字符串的名称是 NET_TOPOLOGY_IMPL_KEY,这是供程序代码用的;字符串的值是"net.topology.impl",这是配置项的名称。所以这个配置项的名称是 net.topology.impl。如前所述,配置项不一定全都出现在配置文件中,程序中也可以在某个具体的 Configuration 对象中动态创建临时的配置项。不过配置文件 core-default.xml 中确实提供了这个名为 net.topology.impl 的配置项:

```
<property>  
  <name>net.topology.impl</name>  
  <value>org.apache.hadoop.net.NetworkTopology</value>  
  <description> The default implementation of NetworkTopology which  
                  is classic three layer one.  
</description>  
</property>
```

这是标准的配置项格式,有名称 name,有值 value,还有说明 description。这里说明采用 NetworkTopology,并说明这是经典的三层结构的拓扑。

对这个配置项是这样使用,别的配置项也是一样,Hadoop 的代码中到处都是这样的运行时动态配置。这个“运行时”配置不一定是初始化阶段进行的配置,那虽然也是“运行时”,但实际上却是半静态的,而 Hadoop 所用的这种方法才真正是动态的。

NetworkTopology 中的内容是怎么来的呢?主要是由集群中各个节点主动向“主节点”报到、登记的。事实上,真正关心拓扑信息的首先是文件系统的主节点,即 HDFS 子系统的 NameNode 节点,因为这个节点需要安排文件内容的存储,这就需要知道哪些节点在同一个机架上,或接在同一台交换机上。另外,YARN 子系统的主节点,即资源管理者 ResourceManager,以及各个具体(应用)作业的管理者,有时候也需要知道集群的拓扑信息。大数据处理的原则是“数据在哪,计算就在哪”,但是有时候因条件所限无法把全部计算或者哪怕是部分计算放在数据所在的那个节点上,从而需要跨节点搬运数据,这时候把计算放在哪些节点上为好呢?这就需要用到拓扑信息了。

可是这里又有个问题。集群中那些运行着 Hadoop 软件的节点,固然可以主动向主节点登记,但是交换机和机架怎么办,有关它们的信息是怎么来的?它们不执行任何 Hadoop 软

件,不会主动登记。所以有关交换机(和机架)的信息有其特殊性,需要特别加以管理。

为此 Hadoop 定义了一个界面 `DNSToSwitchMapping`,意为从 DNS 域名如“x1.y2.com”至交换机节点名的映射。另外还有个界面 `DNSToSwitchMappingWithDependency` 则继承和扩充了这个界面,用于更复杂和灵活的拓扑。定义于 `DNSToSwitchMapping` 界面的函数主要就是 `resolve()`。给定一个(或几个)域名或 IP 地址,如 x1.y2.com,这个函数加以解析并返回该节点所在的位置,如/foo/rack。这里的 foo 是一台交换机,rack 是所在的机架。

抽象类 `AbstractDNSToSwitchMapping` 实现了 `DNSToSwitchMapping` 界面,然而这只是抽象类。实际可用的、对此抽象类的扩充是 `CachedDNSToSwitchMapping`,意为缓存着的映射,但是这依旧没有解决有关交换机和机架的信息来源问题。事实上,在目前 Hadoop 平台的设计中,这些信息来源于人为的配置。Hadoop 的设计中提供了两个方法:一个是通过人工编辑的映射表文件提供信息;另一个是通过脚本生成这些信息。

Hadoop 的代码中有两个类是对 `CachedDNSToSwitchMapping` 的扩充。其一是 `TableMapping`,这是基于映射表文件的;其二是 `ScriptBasedMapping`,这是基于脚本的。至于具体用哪一种,仍是通过配置项设定的。

同样是在 `CommonConfigurationKeysPublic` 这个类中,还有几个常数字符串的定义:

```
String NET_TOPOLOGY_NODE_SWITCH_MAPPING_IMPL_KEY =
    "net.topology.node.switch.mapping.impl"
String NET_TOPOLOGY_SCRIPT_FILE_NAME_KEY = "net.topology.script.file.name"
String NET_TOPOLOGY_TABLE_MAPPING_FILE_KEY = "net.topology.table.file.name"
String NET_DEPENDENCY_SCRIPT_FILE_NAME_KEY =
    "net.topology.dependency.script.file.name"
```

相应地,在配置文件 `core-default.xml` 中,则有:

```
<property>
  <name>net.topology.node.switch.mapping.impl</name>
  <value>org.apache.hadoop.net.ScriptBasedMapping</value>
  <description> The default implementation of the DNSToSwitchMapping. It
    invokes a script specified in net.topology.script.file.name to resolve
    node names. If the value for net.topology.script.file.name is not set, the
    default value of DEFAULT_RACK is returned for all node names.
  </description>
</property>

<property>
  <name>net.topology.script.file.name</name>
  <value></value> //value 为空
  <description> The script name that should be invoked to resolve DNS names to
    NetworkTopology names. Example: the script would take host.foo.bar as an
    argument, and return /rack1 as the output.
  </description>
```



```

</property>

<property>
  <name>net.topology.table.file.name</name>
  <value></value> //value 为空
  <description> The file name for a topology file, which is used when the
    net.topology.node.switch.mapping.impl property is set to
    org.apache.hadoop.net.TableMapping. The file format is a two column text
    file, with columns separated by whitespace. The first column is a DNS or
    IP address and the second column specifies the rack where the address maps.
    If no entry corresponding to a host in the cluster is found, then
    /default-rack is assumed.
  </description>
</property>

```

这里,配置项“net.topology.node.switch.mapping.impl”表明采用 ScriptBasedMapping,因此,就要根据另一配置项“net.topology.script.file.name”确定脚本文件名,但是那个配置项的值为空,表示没有这个脚本。那怎么办呢?这里说了:那样的话,所有节点所在的机架都将是 DEFAULT_RACK,即“/default-rack”。换言之,就是把集群中所有的机器节点都视为同处于同一个机架上,那就是不分机架了。所以,我们说 Hadoop 的 YARN 和 HDFS 子系统都是可以“rack-aware”的,即具有获知具体节点所在机架的能力;但这是有条件的,如果不满足条件就不是“rack-aware”了。

同样,配置项“net.topology.table.file.name”的值也为空,所以即使把前面那个配置项的值换成 TableMapping 也是一样。

有了 NodeBase、NetworkTopology、DNSToSwitchMapping 以后,最好还要有个工具式的模块,把解析节点所在机架的操作及其管理变得更宏观、更方便,Hadoop 代码中的 RackResolver 类就起着这样的作用。下面是这个类的摘要:

```

class RackResolver {}
] static DNSToSwitchMapping dnsToSwitchMapping //把 DNSToSwitchMapping 封装起来
] init(Configuration conf)
  > dnsToSwitchMappingClass = conf.getClass(CommonConfigurationKeysPublic.
    NET_TOPOLOGY_NODE_SWITCH_MAPPING_IMPL_KEY,
    ScriptBasedMapping.class, DNSToSwitchMapping.class)
  > newInstance = ReflectionUtils.newInstance(dnsToSwitchMappingClass, conf)
  > dnsToSwitchMapping = ((newInstance instanceof CachedDNSToSwitchMapping)?
    newInstance : new CachedDNSToSwitchMapping(newInstance))
] resolve(Configuration conf, String hostName) //第一次解析必须调用这个
  > init(conf)
  > return coreResolve(hostName)
] resolve(String hostName) //以后的解析就调用这个

```

```

> coreResolve(hostName)
] coreResolve(String hostName)
> tmpList = new ArrayList<String>(1)
> tmpList.add(hostName)
> rNameList = dnsToSwitchMapping.resolve(tmpList)
> if (rNameList == null || rNameList.get(0) == null) {
>+ rName = NetworkTopology.DEFAULT_RACK
>+ LOG.info("Couldn't resolve " + hostName + ". Falling back to "
+ NetworkTopology.DEFAULT_RACK)
> } else {
>+ rName = rNameList.get(0)
>+ LOG.info("Resolved " + hostName + " to " + rName)
> }
> return new NodeBase(hostName, rName)

```

这个类不提供构造函数,代码中也从不创建 RackResolver 类对象,就相当于一个静态的数据结构和一组函数。其内部成分 DNSToSwitchMapping 其实是实现了这个界面的某类对象,也是作为 static 成分而存在的。这就是说,在一台 Java 虚拟机 JVM 上只会有一个 RackResolver,也并不需要创建,只要有某个 package 导入(import) RackResolver, JVM 就会将其装载进来,但是其内部的 DNSToSwitchMapping 对象(实际上是 ScriptBasedMapping 或 TableMapping 对象)是在其 init() 函数中创建的。需要知道一个节点在什么机架上的时候,就要调用 RackResolver.resolve(hostName) 来加以解析。不过首次调用时一定要用 RackResolver.resolve(conf, hostName), 那样才会调用其 init() 函数以创建 DNSToSwitchMapping 对象。

至于 RackResolver 怎样进行解析,则原理上跟文件路径名(字符串)的解析相仿,文件路径中的最后一个目录节点名就相当于这里的机架节点名,前面的都相当于交换机节点名,读者可以结合着摘要自行阅读源代码,这里就不详述了。

另外, Hadoop 的代码中还定义了一个 Cluster 类。那是 YARN 框架中供各个节点上的 NodeManager 用来跟集群内的其他节点通信的接口。对具体的节点而言,“集群”就是其所处的“世界”,而 Cluster 对象就代表着“境外”,代表着这个节点的“外交”,但是 Cluster 类跟集群的拓扑无关。

3.2 Hadoop 系统的结构

在上述计算机集群的基础上, Hadoop 建立起 HDFS 和 YARN 两个子系统。前者是文件系统,管数据存储;后者是计算框架,管数据处理。

如果只有 HDFS 而没有 YARN,那么 Hadoop 集群可以被用作容错的文件服务器,别的就没有什么应用可言了。虽然 HDFS 是个分布式的文件系统,但是对服务器的用户来说那只是它的内部实现,从外部看与一般基于 Raid 结构的文件服务器并无多大区别。

然而,加上了 YARN 情况就不同了。从功能和层次上看, YARN 是 HDFS 的用户,是

HDFS 的上一层, YARN 的功能和作用是在建立在 HDFS 基础上的, HDFS 提供数据供 YARN 子系统处理和计算。这样, HDFS 的价值通过 YARN 又得到了进一步的体现, 这个价值远远超过了它作为一般文件服务器所具有的价值。与一般游离于文件服务器外部、通过网络(无论其为广域网、局域网或储域网)与服务器相连的应用相比, YARN 代表着一种新的计算模式。在传统的“客户/服务”模式中, 计算在哪里进行, 就把数据(通过网络)搬运到哪里。而 YARN 所实现的模式却是: 数据在哪里, 计算就在哪里进行; YARN 子系统里的计算就好像溶化在 HDFS 所在的集群中一样。传统的“客户/服务”模式是哑铃状的集中存储加集中计算, 而 Hadoop 的模式是“溶化”式的、“打成一片”的分布存储加分布计算。显然, 对于大数据处理, 后者的效率(从而其价值)比前者要大得多。

这两个子系统都是分布式的, 但都是主从式分布。HDFS 子系统中有主节点和从节点, YARN 子系统也有主节点和从节点。主节点管理着从节点, 并对外代表着整个子系统。所谓对外, 主要是对使用人。比方说, 使用人要提交一个进行数据处理的作业(Job), 就只跟 YARN 子系统的主节点打交道, 而不直接跟任何一个从节点打交道; 即使他正在某个从节点的终端上, 也只是借由这个从节点跟主节点打交道。同样, 如果使用人要将一文件复制到 HDFS 中, 也首先要跟 HDFS 的主节点打交道, 然后在主节点的安排和指引下将文件内容传递到某些从节点上。主、从节点之间的关系, 就好像“中央”与“地方”的关系。之所以说主/从, 而不说主/次, 就是要突出这种上下级的管理/被管理的关系。

YARN 和 HDFS 是同一个集群上的两个子系统, 但这并不意味着把集群中的节点机器划分成两个集合, 分别用于两个子系统; 而是集群中的每个节点机器都扮演着两种角色, 打着两份工。所以, 当我们说某台节点机器是 HDFS 的一个从节点时, 它也许(多半)又是 YARN 的从节点, 但也不排除恰巧是 YARN 主节点的可能。

起着 HDFS 主节点作用的是一个 NameNode 对象, 更确切地说是一个运行着 NameNode 类代码的 Java 虚拟机(JVM), 再确切一点说是节点机器上的一个运行着 NameNode 的 JVM 进程。NameNode 类是有主函数 main() 的。同样, 起着 HDFS 从节点作用的是 DataNode 对象, 这也是 JVM 进程, DataNode 类也是有主函数 main() 的。虽然 NameNode 和 DataNode 只是在某个节点机器上运行的对象, 其本身并非物理意义上的节点, 但我们将其当成 HDFS 子系统的逻辑意义上的节点。

起着 YARN 主节点作用的是一个 ResourceManager 对象, 也是一个 JVM 进程; 起着 YARN 从节点作用的是 NodeManager 对象, 也是 JVM 进程。当然, 这两个类都有主函数 main()。同样, 我们也称 ResourceManager 和 NodeManager 对象为节点, 因为它们都是 YARN 子系统的逻辑意义上的节点。

不过, 起着 YARN 主节点作用的是 ResourceManager, 并不意味着其所在的节点机器上只能有这么一个进程属于 YARN 子系统。同样, NodeManager 所在的节点机器上也未必只有一个进程属于 YARN 子系统。无论是 ResourceManager 还是 NodeManager, 都可以再创建别的进程。例如 Mapper 和 Reducer 就都是独立存在的 JVM 进程, 它们都属于 YARN 子系统, 却并非 ResourceManager 或 NodeManager 的内部成分。这样, 假定集群中的某个机器节点在两个子系统中都是从节点, 并且上面又运行着一个 Mapper, 那么这台机器上至少运行着三个独立存在的 Java 虚拟机进程, 分别执行着 NodeManager、DataNode 和 Mapper 三个 class 的程序。当然, 实际的情况还要更复杂。

至于同一进程内部的多个线程,那就更不用说了。

就这样,HDFS的主节点 NameNode 管理着集群中所有的 DataNode,YARN 的主节点 ResourceManager 管理着集群中所有的 NodeManager。但是这二者又有上下层之分,当我们说到一个 Hadoop 集群、Hadoop 平台的时候,更多的是指其 YARN 子系统。我们要在 Hadoop 上运行的应用,更多的是大数据处理而不是大数据存储。而 YARN 子系统的主体,则是其 MapReduce 框架。

YARN 的计算模式是“数据在哪里,计算就在哪里”;但是又说跟文件系统打交道就得跟其主节点 NameNode 打交道。那是否计算就只能在 HDFS 的主节点上进行,或者虽然计算可以在从节点上进行,但所有的数据都得流经主节点呢?当然不是。我们不妨打个比方,把 HDFS 设想成一个大仓库系统,这个系统的仓库散布在全国各地,但是由总部统一管理,账本都在总部,各仓库凭总部开出的提货单发货。现在 YARN 接到一个大型任务(“作业”),要进行某方面的加工提炼。它首先要搞清楚需要一些什么原材料,然后就跟仓储总部联系,搞清楚这些原材料都存放在哪些仓库中,并开好提货单。有了这些,YARN 就可以把加工厂开到一个一个有关仓库的当地,以便就地提货,就地加工,边提货边加工。即使仓库当地挤不下了,也要把加工厂开在尽量靠近的地方,以减少运输开销。另一方面,既然提货单已在手上,就不需要事事都跟总部打交道了。这里的仓储总部相当于 NameNode,一个个具体的仓库就像 DataNode;YARN 需要从 NameNode 得到的只是类似于“介绍信”、“提货单”那样的东西。至于尽量要开到仓库当地的加工厂,那就是 Mapper 和 Reducer,那也都是独立的 JVM 进程。

3.3 Hadoop 的 YARN 框架

如前所述,用户眼中的 Hadoop 更多的是其 YARN 子系统,因为 HDFS 只起存储的作用,YARN 才进行数据的处理和计算。所以,当用户要在 Hadoop 上执行某个“作业(Job)”时,首先要将其提交给 YARN 的主节点 ResouceManager,尽管这个用户很可能是在某个从节点的终端上。从节点只能执行主节点下达的任务,而不能自作主张。

用户在某个从节点将作业提交到主节点后,下面就是主节点如何筹划、调度和指派执行的事了。这里面一个突出的问题就是资源管理,因为怎么调度、把作业指派到哪些节点并监督管理其运行,说到底是个如何分配系统资源的问题。

在早期的 Hadoop 系统,即 Hadoop 2.0 版以前的系统中,还没有 ResouceManager 和 NodeManager,在 Hadoop 集群的主节点上扮演着“中央政府”角色的是 JobTracker,即 JobTracker 类的对象,也是个独立的 Java 虚拟机进程。与此相对,每个从节点上都有个 TaskTracker,也都是独立的 JVM 进程。当然,JobTracker 与 TaskTracker 之间是管理与被管理的关系,所以整个集群在逻辑上是一种单层的主/从式星形结构。JobTracker 把用户提交的作业分解成许多“任务(Task)”,例如一个 Mapper 就是一个 Task,一个 Reducer 也是一个 Task,并根据作业的要求和各节点的资源情况将这些 Task 指派给某些节点,然后由这些节点上的 TaskTracker 将这些 Task 投入运行。当然,主节点也好,从节点也好,将自始至终跟踪管理这些 Task 的运行,只是层次不同,主节点是在作业即 Job 层次上跟踪管理,从节点是在任务即 Task 层次上跟踪管理。Tracker 这个词就是因此而来的。

但是,从 2.0 版开始,Hadoop 引入和实现了一种新的、称为 YARN 的资源管理与调度机

制,形成了相应的 YARN 框架(YARN Framework)。从 Hadoop 2.0 版开始,虽然代码中还留有一些残迹,JobTracker 和 TaskTracker 却已经不复存在,分别被 ResouceManager 和 NodeManager 所取代。

YARN 是“Yet Another Resource Negotiator”的缩写,意为与 JobTracker/TaskTracker 不同的另一种资源协商管理机制。实际上以 YARN 取代 JobTracker/TaskTracker 的努力从 Hadoop 的 0.23 版就开始了,但是这意味着对 Hadoop 代码进行伤筋动骨的改写,所以主流的 Hadoop 版本一时仍维持原来的 JobTracker/TaskTracker 框架不变,直到 2.0 版才正式改用 YARN。

那么采用了 YARN 以后的计算机集群是怎样管理计算资源并调度作业运行的呢?后面有好几章讲的都是 YARN 各方面的细节,但是这里先对 YARN 做一扼要的概述还是很有必要的。

在 YARN 这个框架中,作为一个独立 JVM 进程,在主节点上扮演着相当于“中央政府”角色的是“资源管理者(RM)”,一个 ResourceManager 类对象。在每个从节点(普通节点)上扮演着“地方政府”角色的则是“节点管理者(NM)”,一个 NodeManager 类对象,也是作为一个独立的 JVM 进程而存在的。

主节点上的资源管理者 RM 掌管着集群内各个节点上资源的分配使用,记录着各节点的资源使用情况。这里的所谓资源就是存储器和 CPU。存储器是指虚存空间,CPU 也是指“虚核”,即 VCore。什么叫 VCore 呢?一个进程,一个线程,一台 Java 虚拟机,都可以占用一个 VCore,这要看具体的需要和安排,但是一个线程绝对不需要两个虚核。反过来,多个虚核也可以并发、分时地落实到同一个实体的 CPU 核上,但此时系统的性能就很可能有所下降。所以,一个节点的 VCore 容量某种程度上决定了允许启动 JVM 的数量。

一个作业被提交到 RM,其申请材料首先会被保存起来,排入一个等待调度的队列。然后 RM 通过一个 YARN 调度模块进行调度,首先在它的“账本”中选出一个具有足够空闲资源的节点,让其担任相当于“项目组长”的角色,并将一个操作系统层面的 Shell 命令行连同所分配的资源配额等信息打包作为一个“容器(Container)”发送给这个节点,使其就地启动一个进程作为这个作业的“应用主管”,即 Application Master,缩写成 AM。这就好比,“中央”在收到作业请求后就为其“立项”,并指定一个“地方政府”作为该项目的“牵头单位”,在当地成立一个项目组,组长的角色就称为 AM,就好像企业中把项目经理(Project Manager)称为 PM 一样。

注意,这里说的是“Application Master”,而不是“Job Master”。与从前的 JobTracker 不同,现在主节点上的 RM 已经把视野从“作业”扩展到了“应用”,意思是在 YARN 框架中运行的不再限于原来那样的 Java 作业,也可以是普通的任何“应用”。后面我们将看到,在 YARN 框架中运行的也可以不是 Java 类,还可以不是 MapReduce。

此外,交由“牵头单位”用来启动 AM 进程的 shell 命令行也并非固定不变,而是取决于所提交的具体应用。对于常规的、作为 Java 类的 MapReduce 作业,这个命令行是“~/bin/java ... MRAppMaster ...”,所以 MapReduce 作业的“项目组长”AM 就是 MRAppMaster,意为“MapReduce 的 App Master”。

如上所述,这个 AM 是作为一个独立的进程、独立的 Java 虚拟机运行的,是一个“独立法人”,而不是当地 NodeManager 内部的一个模块,与 NodeManager 不在同一个 JVM 上。

一个应用(作业)一旦有了 AM,这个“项目组长”便要负起对于这个作业的责任。首先要

分析这个作业的种种情况(在一个称为 Context 的数据结构中),看需要把它分解成几个任务,有几个任务就需要几个容器,即打成几个包。假定一个作业要求有 16 个 Mapper 和一个 Reducer,那么至少就需要 17 个容器(不算 AM 本身所占的容器),或者说需要打成 17 个包,这样才能把它们分发到不多于 17 个的节点上去运行,这就好像包工头把一个项目拆散了分包出去一样。注意,“容器(Container)”并非物理的概念,实际上只是对用来完成具体任务的软件(一般是 Java 类)及其所分配资源和所指派节点的描述,但是到了所指派的节点上就会落实为一个独立的进程。如果是 Java 程序,那就是一个独立的 JVM。至于其所需的资源,则如前所述就是内存空间和虚拟核 VCore 的数量。

容器是要向 RM 申请的,并非 AM 可以自行发放,因为 RM 统管着全局的资源。所以,AM 需要就具体应用所含的任务逐一向 RM 申请容器。而 RM 则通过其调度员(Scheduler)加以调度,根据当时的资源情况和具体任务的需求分配资源并将此任务指派到某个节点,这样就形成了一个容器。不过 RM 并不直接将容器发送给所指派的节点,而只是发回给 AM。一级管一级的事,RM 只管到这一级,下面就是 AM 的事了。

当然,调度的时候可以有各种不同的考虑和策略,所以 Hadoop 提供了三个不同的 Scheduler,以插件的形式供选择,你想用哪一种就插上哪一种。这三种 Scheduler 是 FifoScheduler、FairScheduler 和 CapacityScheduler。其中 FifoScheduler 是最简单的,它的调度策略是 FIFO,先来先得,只要手里的资源还够,就给。当然,同一个容器中的资源,即同一个任务所需的资源,必须是在同一个节点上。FairScheduler 就要复杂一些,其调度策略是 Fair,要考虑作业之间,实际上是用户之间的公平合理。CapacityScheduler 就更麻烦,还要考虑怎样使用资源才可在总体上臻于最优。如果不另行规定,那么默认的就是最复杂的 CapacityScheduler。

AM 得到了 RM 分配下来的容器之后,就要与该容器所指派的节点联系,将容器转发给这个节点,以“发动(launch)”这个容器所包含的任务在目标节点上运行。而受指派的节点,在拿到一个容器之后,则先要把容器中指定的数据文件(或片断)和程序映像“本地化(localize)”,将它们复制到本地,然后就启动这个任务,作为一个进程运行。

为什么要把容器中的任务作为进程运行,(对于 Java 类)为什么要为其启动一个独立的 JVM,而不是放在 NodeManager 所在的 JVM 上作为一个模块(“类”)运行呢? 这里有几个原因。首先是对安全的考虑。假定这个任务是一个 Mapper,这一般是由用户提供的,程序质量得不到保证,里面万一有个什么 Bug 就可能把整个 NodeManager 拖垮。而若是作为一个独立的进程运行就不怕,到时候大不了把这进程杀(kill)了。其次还有灵活性的考虑。在新采用的 YARN 框架中,具体任务所要执行的也不一定就是 Java 程序,也可能是个 C/C++ 的应用,甚至还可以是个脚本,那就难以纳入 NodeManager。再说,在一个节点上可能需要运行分属不同应用的多个任务,例如可能有来自应用 A 的两份 Mapper 和应用 B 的一份 Mapper 加一份 Reducer,如果不把它们作为独立的进程运行,就会给管理带来困难。

发动了容器所含的任务在所指派节点上运行之后,AM 和 RM 并非就此袖手旁观,而是通过“心跳(Heartbeat)”密切注视和管理着这些任务的运行。假定 AM 发现自己在某个节点上的任务出了问题(例如超时没有心跳),并且事先已安排了备用容器(称为 Speculator),它就可能下达命令,把出问题的进程 kill 掉,并让备用容器(通常是在另一个节点上)顶上去。否则,如果事先没有安排备用容器,它也可以临时向 RM 另外申请一个容器,以这个新分配的容

器替换已经出了问题的容器。此外,用户常常要查询作业的进展情况,但是用户并不知道自己这个作业的 AM 在哪里,也无法直接与 AM 交互,因此所有的查询都要流经 RM。

这样的方案,这样的机制和框架,应该说是很合理的,读者不妨想想是否还有更合理的方案。读者也许会问,为什么要把 AM 放到别的节点上,而不是留在主节点上,甚至就作为 RM 的内部模块加以执行?其实早期的 JobTracker 正是这样的。但是想想:要是集群的规模达到了几千台机器,甚至上万台机器,那么同时在集群中运行的应用数量,也即 AM 的数量就可能很大,这对于主节点显然是个不小的负担。另一方面,如果把 AM 都集中在主节点上,就会把本可以分散的许多网络流量都集中到中央而造成拥堵。最后,把 AM 都集中在一个节点上,也使单点故障的后果变得更加突出而难以恢复。

当然,这么一来对于作业的管理就复杂起来了,系统的开销也因此而增加了不少。然而事情往往就是这样,一种计算模式,如果只是小规模应用,固然可以做得很简洁很高效,而如果想要对付大规模应用,就免不了一定程度的复杂性上升和效率下降。

3.4 状态机

如前所述,YARN 主节点 RM 受理用户提交的作业之后要为其创建一个 AppManager,即 AM,还要为其分配资源,然后要密切注视作业的进展,最后还要处理善后。这显然需要在主节点上维持关于这个作业的上下文,因为作业有它自己的生命周期,从其产生到推进,再到消亡,是一个过程,在此过程中作业的状态在改变,主节点上需要有些与此相适应的操作。通常,像这样得到受理并且正在推进中的作业不是只有一个而是有很多个。显然这些作业的生命周期有时间上的重叠,因而存在着并发。也就是说,RM 管理着诸多并发的作业。对于并发的过程,如果过程复杂而粒度细小,就只能用线程或进程的方法加以实现。但是,对于相对简单并且粒度较粗的过程,则也可以采用一种简单一些的实现方法,那就是状态机。而 RM 对于作业的管理,就比较适合采用状态机的方法。

这样的情况在 YARN 中也不只是 RM 一处。以 AM 为例,它就管理着属于同一个作业的诸多“任务(task)”,这些任务分布在许多节点上,有着大致相同的生命周期,所以每个 AM 都管理着诸多并行或并发的任务。注意不要混淆,这些任务本来就是作为独立的 JVM 进程在运行的,这里说的是 AM 对这些任务的管理,例如发现某个任务停滞不前了就要安排后备替换,任务运行结束了就要安排善后事宜,等等。再例如,即使就某个具体的 Mapper 任务而言,也需要先进行资源本地化,可能需要从若干不同的节点复制程序映像和数据,这又是一些并发的过程。

所以 YARN 的代码中广泛使用着状态机(这也是 2.0 版前后的一个显著区别),我们有必要先分析一下这些状态机的实现,以期对这些状态机的作用有比较深入的理解。

这里所讲的“状态机(state machine)”,当然是对“有限状态机”的简称。状态机其实也是一种并发机制,试想如果一个过程可以从开头一直走到结束,而不是“走走停停”,那就没有必要用状态机了。反过来,如果每一个过程都是通过一个独立的进程或线程加以执行,那么这个进程或线程本身就相当于一个(状态数量极多且粒度极小的)状态机。所以,在某种意义上,可以把进程和线程看成由操作系统运行的状态机。但是,如果要由应用层软件去管理多个这样的过程,而且过程又不太复杂,粒度也比较粗,那么为每个过程建立一个状态机也是不坏的选择。

在状态机模型中,一个宏观的过程被抽象成一台机器,其结构包括一组“状态”、一组触发规则和一组操作。只要不是正在处理某个事件的过程中(这是微观的过程),这台机器就总是停留在其中的某个状态上。然后,如果有什么事件发生,这台机器就受到“触发”,这时候它就要查一下规则,看看在当前的状态下遇到这样的事件是否需要处理一下,做点什么,以及是否需要跳变到另一个状态,如果需要就加以执行。状态机对于事件的处理,包括状态的跳变,是“原子”的,即不可分割的。这就是说,如果状态机在“做点什么”的过程中又来了一个新的事件,那么暂时就不加理会,要到做完了对本次事件的处理并完成状态跳变(如果需要的话)以后才又可受到触发。这就像操作系统不允许中断服务嵌套一样。

通常在讲述状态机的书上都会把状态机画成图,图上的“节点”代表状态,而带箭头的“弧”则代表跳转规则,即在什么事件的触发下会做些什么并跳转到什么节点(状态)。这对于简单的状态机倒挺合适,可是对于比较复杂的状态机就不太合适了,因为那样的状态图看上去可能会像是一团乱麻,还不如看着跳转表更觉清晰。

从程序设计的角度看,定义一组状态是简单的,那就是一个枚举(enum)类型,定义一组事件也一样。而跳转规则就麻烦一些,但也并不复杂,因为那就是一个表,或者说一个结构数组。表的每一行代表一条规则,如果定义了 M 种状态和 N 种事件,那么理论上这个表就应该有 $(M \times N)$ 行,但是在具体实现时有些行或许可以合并。一般而言,在一行中应该有 4 个字段,或者说这个表应该有 4 列,那就是当前状态、(到来的)事件类型、下一状态和所需的操作处理。至于里面具体的内容,那就是具体流程的问题,而不是状态机这种模型和机制的问题了。

最后,怎样使用这张跳转表,怎样完成所规定的操作和跳变,这就都需要由程序来实现了,其中之一称为 Dispatcher。Dispatch 本是“派遣”和去向什么地方的意思,发生了一个什么事件时,是 Dispatcher 根据这事件的类型和参数确定应该用来驱动哪一个状态机,并将其交给这状态机的“引擎”。

在 Hadoop 的代码中,具体的状态机,特别是其跳转表,都是由程序动态生成的,而不是静态预定的。用来生成状态机及其跳转表的 Java 类是 StateMachineFactory,即“状态机工厂”。之所以是这样,以作业为例,Hadoop 系统中可能有很多个作业,每个作业都得有自己的状态机,这些状态机有相同的跳转表(因而有相同的状态集合、相同的触发事件集合和操作集合),但是每个状态机所处的状态可能不同。显然,两个作业不能共享同一个状态机,但是却可以共享同一个状态机工厂,因为它们的跳转表是一样的。再说,我们无法事先估计作业的数量,而只能来一个作业就为其动态“生产”一个状态机。下面是 StateMachineFactory 这个类的摘要。

```
class StateMachineFactory< OPERAND, STATE extends Enum<STATE>,
    EVENTTYPE extends Enum<EVENTTYPE>, EVENT >{
    //这是模版(Template)式类型定义,参数类型不同就有不同的 StateMachineFactory
    ] TransitionsListNode transitionsListNode
    ] Map<STATE,
        Map<EVENTTYPE, Transition<OPERAND, STATE, EVENTTYPE, EVENT>>>>
    stateMachineTable           //状态机的跳转总表,表中对于每个状态都有一个子表
    ] STATE defaultInitialState //状态机的默认初始状态

    ] class TransitionsListNode{}
```

```

] class ApplicableSingleOrMultipleTransition<...>{}
] class SingleInternalArc implements Transition<...> {}
] class MultipleInternalArc implements Transition<...> {} //与 SingleInternalArc 类似
] class ApplicableSingleOrMultipleTransition<...> implements ApplicableTransition<...>{}
] class InternalStateMachine{
]] OPERAND operand
]] STATE currentState
]] doTransition(EVENTTYPE eventType, EVENT event)
    > currentState = StateMachineFactory.this.doTransition(operand, currentState,
                                                                eventType, event)

```

对 StateMachineFactory 类的定义是个模版 (Template) 定义。任何状态机工厂都是针对特定四个要素的, 那就是 < OPERAND, STATE, EVENTTYPE, EVENT >。这里 OPERAND 是状态机拥有者的类型, 说明具体的状态机是为谁所用。STATE 说明这台状态机有些什么状态, 这通常是一个枚举类型。EVENTTYPE 是用于这台状态机的事件类型, 通常也是一个枚举类型。而 EVENT, 则是具体事件所属的 class。只要有其中任何一个要素的类型不同, 那就是一种不同的状态机工厂, 从而生产出来的状态机也就不同。不过这种不同只是对象类型上的不同, 就好比只是所加工的材料不同, 工厂及其产品 (在这里就是状态机) 的结构和形态都是一样的。那么这些要素的类型到什么时候才确定呢? 这要到创建具体的 StateMachineFactory 对象的时候。我们不妨看两个例子:

```

class JobImpl implements ...{}
] StateMachineFactory<JobImpl, JobStateInternal, JobEventType, JobEvent>
                                                                stateMachineFactory

class LocalizedResource implements ...{}
] StateMachineFactory<LocalizedResource, ResourceState,
                                                                ResourceEventType, ResourceEvent> stateMachineFactory
//对于 LocalizedResource, EVENTTYPE 是 ResourceEventType, 而 EVENT 是 ResourceEvent

```

在 JobImpl 和 LocalizedResource 内部都有 StateMachineFactory, 这两个状态机工厂的结构相同, 但是因为具体的用途不同, 两个状态机工厂的类型是不一样的。模版定义中的 OPERAND, 在 JobImpl.stateMachineFactory 中是个 JobImpl 类的对象, 而在 LocalizedResource.stateMachineFactory 中则是个 LocalizedResource 类对象。模版定义中的 STATE, 对于前者是为 JobImpl 的状态机定义的一组状态 (通常是枚举类型), 对于后者则是为 LocalizedResource 的状态机定义的一组状态 (另一个枚举类型)。余可类推。

但是这些要素类型的不同并不妨碍二者同为 StateMachineFactory, 因为它们内部的结构成分相同, 提供的操作方法也相同。

StateMachineFactory 类最重要的内部数据成分是 stateMachineTable, 这是一个 MAP, 就是映射表, 或者说是便查表, 是若干二元组的集合。这个 MAP 的类型定义也是模板型定义, 而且更复杂, 这里要加一点解释。

首先, stateMachineTable 是一个映射表, 表中的每个元素都是一个二元组, 让你给定一个 STATE 就可得到它的映射。但是, 如上所述 STATE 的具体类型可以不同, 它所映射的目标的类型也可以不同, 所以 MAP 的类型定义也是模板型定义。我们可以想象, 资源管理者 RM 所用状态机中的状态类型, 与资源本地化过程所用状态机中的状态类型, 理所当然是不一样的, 可是因此就要定义许多不同的 MAP, 那就太麻烦了, 这就是模版的意义所在。那么这里 STATE 的映射是什么呢? 是另一种 MAP, 那也是一个映射表, 是从 EVENTTYPE 到 Transition 的映射表, 让你可以根据事件类型查到相应的跳变说明。而 Transition 则是关于一次具体跳变的说明, 其内容包括伴随着跳变需要执行的操作, 也即对于事件的反应, 以及状态机在跳变后所处的状态。

这样, 如果我们把第一层的 MAP 即 stateMachineTable 展开, 并把 MAP 和 Transition 的内容用方括号框起来, 那么就是这个样子:

```
MAP stateMachineTable [
    STATE, MAP transitionMap [
        EVENTTYPE, Transition
        EVENTTYPE, Transition
        ... //别的事件类型, 别的跳变
    ]
    STATE, MAP transitionMap [
        ...
    ]
    ... //别的当前状态, 别的 transitionMap
]
```

可以这样理解, stateMachineTable 是个二列表, 表中的每一行都是个 STATE 和 MAP 的二元组, 状态机中定义了多少种状态, 这个表中就有几行。这个 STATE 代表着状态机的当前状态, 每个不同的状态都决定了一个第二层的 MAP, 这里称之为 transitionMap, 代表着一组跳变规则。然后, 这个第二层的 MAP 又是一个二列表, 这个表中的每行都是 EVENTTYPE 和 Transition 的二元组, 使每个事件类型都对应着一种跳变。这种对应是有条件的, 只有当状态机处于当前这种状态时, 这样的事件才对应着这样的跳变。如果是在另一种当前状态, 同样的事件就可能对应着另一种跳变了。

于是, 如果发生了针对某状态机的事件, 就可以在该状态机的 stateMachineTable 中找到其当前状态所对应的 transitionMap; 再根据事件类型在该 transitionMap 中找到相应的跳变说明 Transition, 该 Transition 给出了需要执行的操作以及跳变后的状态。

一个 Transition 描述了一种跳变, 其构成要素有二: 一是伴随着跳变的操作; 二是跳变后的状态。但是, 针对不同的状态机, 虽然同是关于跳变的描述, 并且结构相同, Transition 的类型定义却涉及四个因素的类型差异, 所以 Transition 的类型也是模版形式的 “Transition<OPERAND, STATE, EVENTTYPE, EVENT>”。要理解为什么涉及四种因素的类型差异, 我们不妨通过几个实例看看 Transition 的定义和实现。首先, Transition 不是一个类, 而是一个界面, 所以凡是程序代码中用到 Transition 的其实都是指实现了此种界面的某类对象。


```
interface Transition<OPERAND, STATE extends Enum<STATE>,
    EVENTTYPE extends Enum<EVENTTYPE>, EVENT> {
    STATE doTransition(OPERAND operand, STATE oldState,
        EVENT event, EVENTTYPE eventType);
}
```

这个界面只定义了一个方法函数 `doTransition()`，这个函数有四个参数，这些参数的类型可以因不同的状态机而不同，因而这四者就都出现在 `Transition` 的模版式定义中，并且也出现在前面 `StateMachineFactory` 的类型定义中。

那么有哪些类是实现了这个界面的呢？有两个，都定义在 `StateMachineFactory` 内部：

```
class StateMachineFactory<...> {}
] class SingleInternalArc implements Transition<OPERAND, STATE,
    EVENTTYPE, EVENT>{}

]] STATE postState
]] SingleArcTransition<OPERAND, EVENT> hook; // transition hook,操作挂钩
]] doTransition(OPERAND operand, STATE oldState, EVENT event, EVENTTYPE eventType)
    > if (hook != null) hook.transition(operand, event)
        //通过操作挂钩执行该跳变的 transition()函数
    > return postState
] class MultipleInternalArc implements Transition<OPERAND, STATE,
    EVENTTYPE, EVENT>{}

]] Set<STATE> validPostStates
]] MultipleArcTransition<OPERAND, EVENT, STATE> hook; // transition hook,操作挂钩
]] doTransition(OPERAND operand, STATE oldState, EVENT event, EVENTTYPE eventType)
    > postState = hook.transition(operand, event) //通过操作挂钩执行跳变的 transition()函数
    > if (!validPostStates.contains(postState)) {
    >+ InvalidStateTransitonException(oldState, eventType)
    > }
    > return postState
```

`StateMachineFactory` 内部定义了 `SingleInternalArc` 和 `MultipleInternalArc` 这两种“弧 (Arc)”的类型。弧在状态图中代表着跳变，所定义的这两种弧都实现了 `Transition` 界面，都提供一个 `doTransition()` 函数。二者的区别是，`SingleInternalArc` 是从单个 `oldState` 到单个 `postState` 的跳变，而 `MultipleInternalArc` 可以从同一个 `oldState` 到多个 `postState` 之一的跳变，具体取决于执行 `hook.transition()` 的结果。

以 `SingleInternalArc` 为例，可想而知其内部成分 `postState` 和 `hook` 都是在构造函数中得到设置的。其中 `postState` 是跳变后的状态；而 `hook` 意为操作挂钩，其实是一个 `SingleArcTransition` 对象，`doTransition()` 调用 `hook.transition()`，就是调用由此对象提供的 `transition()` 函数。

从上面这段代码摘要中可见，作为类型模版，`Transition` 对于 `OPERAND` 等四个因素的依赖跟 `doTransition()` 的参数类型一致，也跟 `StateMachineFactory` 所依赖的那四个类型一

致。我们在前面已经通过实例看了在实际的(而不是模版的)StateMachineFactory 类型定义中这些抽象类型究竟可以是什么。

到现在为止,我们还只是讲了 StateMachineFactory 的数据部分和类型定义部分,还没有讲到它的操作部分,所以还要看一下 StateMachineFactory 操作方法部分的摘要:

```
class StateMachineFactory<...> {}
] StateMachineFactory(STATE defaultInitialState) //构造函数之一
    > this.transitionsListNode = null
    > this.defaultInitialState = defaultInitialState
    > this.optimized = false
    > this.stateMachineTable = null
] StateMachineFactory( //构造函数之二
    StateMachineFactory<OPERAND, STATE, EVENTTYPE, EVENT> that,
    ApplicableTransition<OPERAND, STATE, EVENTTYPE, EVENT> t)
    > this.defaultInitialState = that.defaultInitialState
    > this.transitionsListNode = new TransitionsListNode(t, that.transitionsListNode)
    > this.optimized = false
    > this.stateMachineTable = null
] StateMachineFactory(...) //构造函数之三
] makeStateMachineTable() //生成跳转表
] addTransition(...) //在当前 StateMachineFactory 的基础上添加单弧跳转规则
    > s = new SingleInternalArc(postState, hook) //创建一个指明目标状态和伴随操作的单弧
    > a = new ApplicableSingleOrMultipleTransition<...>(preState, eventType, s)
    //创建包含此单弧的跳变规则
    > new StateMachineFactory<...>(this, a) //创建加上该规则的新 StateMachineFactory
] addTransition(...) //在当前 StateMachineFactory 的基础上添加多弧跳转规则
] installTopology()
    > return new StateMachineFactory<OPERAND, STATE, EVENTTYPE, EVENT>(this, true)
    >> this.defaultInitialState = that.defaultInitialState
    >> this.transitionsListNode = that.transitionsListNode
    >> this.optimized = optimized
    >> if (optimized) makeStateMachineTable()
    >> else stateMachineTable = null
] doTransition(OPERAND operand, STATE oldState, EVENTTYPE eventType, EVENT event)
    > Map<...> transitionMap = stateMachineTable.get(oldState)
    //根据当前状态获取 transitionMap
    > if (transitionMap != null) {
    >> Transition<...> transition = transitionMap.get(eventType)
    //根据事件类型获取跳转规则,通常是一个 SingleInternalArc 对象
    >> if (transition != null) return transition.doTransition(operand, oldState, event, eventType)
    //调用该跳转规则的 doTransition()方法
```

```

> }
] make(OPERAND operand, STATE initialState) //生成一台针对具体应用的状态机
> new InternalStateMachine(operand, initialState)
>> this.operand = operand
>> this.currentState = initialState
>> if (!optimized) maybeMakeStateMachineTable()
>>> if (stateMachineTable == null) makeStateMachineTable()
] generateStateGraph(String name) //生成代表着状态机的状态图

```

凡是要使用状态机的模块(class),首先必须要调用 StateMachineFactory 的构造函数之一,创建一个基本空白的 StateMachineFactory 对象,然后通过 addTransition()一条条添加跳变规则。每次调用 addTransition()时,都会通过上面的构造函数之二在原有的 StateMachineFactory 对象的基础上创建并返回一个新的 StateMachineFactory 对象。所以 StateMachineFactory 对象是 immutable。同样,StateMachineFactory 内部的 TransitionsListNode 也是 immutable,每次调用 addTransition()就在原有基础上创建一个新的 TransitionsListNode,而所添加的跳变规则就积累在 TransitionsListNode 中。最后,完成了规则的添加之后,就调用 installTopology(),这个函数又调用 StateMachineFactory 的构造函数之三,这次就会调用 makeStateMachineTable(),根据 TransitionsListNode 的内容创建跳转表。注意,这样形成的是状态机工厂 StateMachineFactory,而不是状态机 InternalStateMachine,所以最后还得调用这里的 make()方法创建状态机。创建状态机时在 InternalStateMachine 的构造函数中也可以构建跳转表。当然,跳转表只需构建一次,要么在创建状态机工厂时构建,要么在创建状态机时构建,这取决于 StateMachineFactory 内部的一个变量 optimized,如果未经调用 installTopology(),这个变量就是 false,那就在通过 make()创建状态机的时候再创建。

源代码中涉及很多模板式即“泛型”的定义和引用,做摘要的时候不得不用省略号取代,否则就太长了,读者不应只满足于阅读摘要,应该结合着摘要去看源代码。

光看 StateMachineFactory 的摘要或代码也许还是不甚了了,最好还要结合实际的状态机看一下。为此我们以 Hadoop 代码中最简单的状态机,即 LocalizedResource 的状态机作为实例来做进一步的讲解分析。

一个 LocalizedResource 对象代表着一份需要本地化的具体资源,这里所谓的“资源”是指数据文件或可执行程序。每一份这样的资源都只有四个状态,即 INIT、DOWNLOADING、LOCALIZED、FAILED。这些状态的意义不言自明,因为所谓“本地化”就是去别的节点下载,而下载可以成功也可能失败。定义于资源本地化的事件则有五种:REQUEST、LOCALIZED、RELEASE、LOCALIZATION_FAILED 和 RECOVERED。

我们先看 LocalizedResource 对象的创建,当然这个类有它的构造函数 LocalizedResource()。但是,如果一个类的内部定义了某些结构成分并且有赋值,那么在具体创建一个对象时首先执行的是这些结构成分的赋值,然后才是执行其构造函数。特别地,如果一个类的内部有静态成分,那么更是在创建该类的第一个对象之初就先创建这些静态成分。这些静态成分供该类的所有对象共享,以后再创建该类对象时就可直接加以引用。

下面是 LocalizedResource 类定义的摘要：

```
class LocalizedResource implements EventHandler<ResourceEvent> {}
] StateMachine<ResourceState, ResourceEventType, ResourceEvent> stateMachine
] static StateMachineFactory<LocalizedResource, ResourceState,
    ResourceEventType, ResourceEvent> stateMachineFactory =
    new StateMachineFactory<LocalizedResource, ResourceState,
        ResourceEventType, ResourceEvent>(ResourceState.INIT)
    .addTransition(ResourceState.INIT, ResourceState.DOWNLOADING,
        ResourceEventType.REQUEST, new FetchResourceTransition())
    ...installTopology() //添加了很多跳变规则之后调用 installTopology()
] LocalizedResource(LocalResourceRequest rsrc, Dispatcher dispatcher)
    > this.rsrc = rsrc
    > this.dispatcher = dispatcher
    > this.ref = new LinkedList<ContainerId>()
    > this.stateMachine = stateMachineFactory.make(this)
] ...
```

LocalizedResource 内部有两个与状态机有关的结构成分,其中之一是 stateMachine,这是实现了界面 StateMachine 的某类对象,至于究竟是什么类型我们后面会看到;另一个就是 StateMachineFactory 类对象 stateMachineFactory。这二者的类型定义都是模版式的,因为是用于资源本地化,所以 OPERAND、STATE 等四个抽象类型(泛型)在这里落实为 LocalizedResource、ResourceState、ResourceEventType 和 ResourceEvent。

先看 stateMachine,它既非静态变量,也并未赋值,所以在创建 LocalizedResource 对象之时这个成分的值为 null。

再看 stateMachineFactory,这是个静态成分,而且又有赋值,我们把前面摘要中省略的语句全文列出于下:

```
private static final StateMachineFactory
    <LocalizedResource, ResourceState, ResourceEventType, ResourceEvent>
stateMachineFactory =
    new StateMachineFactory<LocalizedResource, ResourceState,
        ResourceEventType, ResourceEvent>(ResourceState.INIT)
        //初始状态为 INIT,下面一条条添加跳变规则
    .addTransition(ResourceState.INIT, ResourceState.DOWNLOADING,
        ResourceEventType.REQUEST, new FetchResourceTransition())
    .addTransition(ResourceState.INIT, ResourceState.LOCALIZED,
        ResourceEventType.RECOVERED, new RecoveredTransition())

    // From DOWNLOADING (ref > 0, may be localizing)
    .addTransition(ResourceState.DOWNLOADING, ResourceState.DOWNLOADING,
        ResourceEventType.REQUEST, new FetchResourceTransition())
```

```

// TODO: Duplicate addition!!
.addTransition(ResourceState.DOWNLOADING, ResourceState.LOCALIZED,
    ResourceEventType.LOCALIZED,
    new FetchSuccessTransition())
.addTransition(ResourceState.DOWNLOADING, ResourceState.DOWNLOADING,
    ResourceEventType.RELEASE, new ReleaseTransition())
.addTransition(ResourceState.DOWNLOADING, ResourceState.FAILED,
    ResourceEventType.LOCALIZATION_FAILED, new FetchFailedTransition())

// From LOCALIZED (ref >= 0, on disk)
.addTransition(ResourceState.LOCALIZED, ResourceState.LOCALIZED,
    ResourceEventType.REQUEST, new LocalizedResourceTransition())
.addTransition(ResourceState.LOCALIZED, ResourceState.LOCALIZED,
    ResourceEventType.RELEASE, new ReleaseTransition())
.installTopology();

```

注意,这里从头到尾只是一个语句,一个很长、看起来似乎有点怪异的语句,Hadoop 代码中的状态机都是这样创建的。这个语句先调用 `StateMachineFactory` 类的构造方法,创建一个空白的 `StateMachineFactory` 对象,然后调用 `StateMachineFactory.addTransition()`,在其跳转表中添加跳转规则。如前所述,`addTransition()` 所返回的仍是 `StateMachineFactory`,于是就可以一条一条往下加,直到最后调用 `installTopology()`,那仍旧还是返回 `StateMachineFactory`,因而最后可以赋值给变量 `stateMachineFactory`。

如果把跳转表的内容画成状态图,那么每增加一条跳转规则就相当于在图上增添了一条表示跳转的弧。除起点和终点之外,与一次跳转相联系的还有触发事件和作为响应的操作。这是 `addTransition()` 的调用界面:

```
[StateMachineFactory.addTransition()]
```

```
StateMachineFactory<OPERAND, STATE, EVENTTYPE, EVENT>
```

```
addTransition (STATE preState, STATE postState, EVENTTYPE eventType,
    SingleArcTransition<OPERAND, EVENT> hook)
```

首先这个函数的返回值仍是一个 `StateMachineFactory`。这很好理解:给一个状态机工厂增添一条跳变规则,所得到的仍是一个状态机工厂。这看似是在为状态机工厂增添规则,实际上这些规则都会出现在由此工厂生产的状态机中,所以这实质上是在为将来要生产创建的状态机增添规则。函数的调用参数是四个: `preState` 和 `postState` 是跳变前后的状态; `eventType` 是引起跳变的事件类型,更确切地说,是当状态机处于 `preState` 状态的条件下可以引发去往 `postState` 状态的事件类型;最后一个参数 `hook` 给定了伴随着跳变的操作提供者。

参数 `hook` 的类型是 `SingleArcTransition`,但这只是个 `interface`,所以应该是个实现了这个界面的某种类型。其实 `SingleArcTransition` 这个界面只定义了一种操作 `transition()`,这就是伴随着跳变的操作:


```
public interface SingleArcTransition<OPERAND, EVENT> {
    public void transition(OPERAND operand, EVENT event)
}
```

所以参数 hook 应该是实现了这个界面,即提供 transition()操作方法的某类对象。注意,transition()这两个参数的类型都是模板中使用的抽象类型,在这里落实为 LocalizedResource 和 ResourceEvent。这里要说明,函数 transition()只是伴随着跳变的操作,而不是跳变本身。状态机并非因为执行了这个函数才发生跳变,而是因为状态机对于跳变规则的执行。

那么实现了 SingleArcTransition 界面的类究竟是什么呢?这有很多,具体到资源本地化的状态机,我们以前面代码中的最后一次 addTransition()调用为例:

```
addTransition(ResourceState.LOCALIZED, ResourceState.LOCALIZED,
               ResourceEventType.RELEASE, new ReleaseTransition())
```

这次调用所增添的规则是这样的:如果状态机的当前状态 preState 是 LOCALIZED,发生了类型为 ResourceEventType.RELEASE 的事件,则发生以 LOCALIZED 为目标的跳变,所以目标状态 postState 仍保持 LOCALIZED 不变,跳变时执行由一个 ReleaseTransition 对象所提供的操作,那就是 ReleaseTransition.transition()。

那么 ReleaseTransition 是个实现了 SingleArcTransition 界面的类吗?是的:

```
class LocalizedResource implements EventHandler<ResourceEvent> {}
] class ReleaseTransition extends ResourceTransition {}
]] void transition(LocalizedResource rsrc, ResourceEvent event)
```

此刻我们还不关心 LocalizedResource.ReleaseTransition.transition()究竟做些什么,但是我们看到 ReleaseTransition 是定义于 LocalizedResource 内部的一个类,这个类是对 ResourceTransition 的扩充,而 ResourceTransition 则实现了 SingleArcTransition 界面。实际上 ResourceTransition 是个抽象类,真正实现了 SingleArcTransition 界面并提供 transition()函数的,在这里就是 ReleaseTransition。所以上述 addTransition()调用的最后一个参数 hook 就是临时通过 new 操作创建的 ReleaseTransition 对象。

同样的道理,前面的第一次 addTransition()调用所添加的规则是:如果当前状态为 INIT,而且发生了事件 ResourceEventType.REQUEST,就执行 FetchResourceTransition.transition(),并跳转到状态 DOWNLOADING,因而状态机的当前状态就从 INIT 变成 DOWNLOADING。

这样,在创建一个 LocalizedResource 对象的时候,在调用其构造方法之前就已经把它的 StateMachineFactory,包括其跳转表准备好了。当然,凡是像 LocalizedResource 这样需要创建状态机的类,都得通过程序代码往跳转表中放上自己的跳变规则,以形成自己的 StateMachineFactory。需要本地化的资源可能会有很多,每项资源都要有个状态机,这些状态机都将有相同的跳转表。

还要说明,这里的 stateMachineFactory 是 LocalizedResource 类的静态成分,那是由整个类所共享的,但 stateMachine 却不是,每个具体的 LocalizedResource 对象都需要创建自己的状态机 stateMachine。这当然很好理解,资源的本地化本来就是每项具体资源的事,是互不相干的事。由此可知,虽然 StateMachineFactory 为数不多,但是状态机的数量可以很大。

注意,至此为止只是状态机工厂的创建,这是在调用 `LocalizedResource` 类的构造函数之前发生的,这里还没有涉及状态机的创建。

完成了内部数据成分的初始化之后,就要调用 `LocalizedResource` 的构造函数了:

```
public LocalizedResource(LocalResourceRequest rsrc, Dispatcher dispatcher) {
    this.rsrc = rsrc;           //这是具体的资源请求
    this.dispatcher = dispatcher; //准备用于这个状态机的 Dispatcher
    this.ref = new LinkedList<ContainerId>();
    ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
    this.readLock = readWriteLock.readLock(); //用来保护并发读操作的锁
    this.writeLock = readWriteLock.writeLock(); //用来保护并发写操作的锁

    this.stateMachine = stateMachineFactory.make(this); //生成状态机
}
```

这里我们关心的是最后一个语句,即状态机的生成。注意调用参数 `this`,这代表着正在构造的 `LocalizedResource` 对象,即一项具体的需要加以本地化的资源。凡是这个对象内部的结构成分,例如 `this.rsrc`、`this.dispatcher`,包括 `this.readLock` 和 `this.writeLock`,都会随同用作参数的 `this` 被传递给 `stateMachineFactory.make()`。换言之,在 `stateMachineFactory.make()` 内部是可以访问到这些结构成分的。

我们接着看 `stateMachineFactory.make()` 的代码。

```
[LocalizedResource.LocalizedResource() > StateMachineFactory.make()]
```

```
public StateMachine<STATE, EVENTTYPE, EVENT> make(OPERAND operand) {
    return new InternalStateMachine(operand, defaultInitialState);
}
```

可见实际生成的是个 `InternalStateMachine` 类的对象。`InternalStateMachine` 是个定义于 `StateMachineFactory` 内部的类,这个类实现了 `StateMachine` 界面,所以我们可以把返回的 `InternalStateMachine` 对象赋值给 `LocalizedResource.stateMachine`。

下面是 `InternalStateMachine` 类构造函数的摘要:

```
[LocalizedResource.LocalizedResource() > StateMachineFactory.make()
> InternalStateMachine.InternalStateMachine()]
```

```
InternalStateMachine(OPERAND operand, STATE initialState)
```

```
> this.operand = operand
```

```
> this.currentState = initialState
```

```
> if (!optimized) {
```

```
>+ maybeMakeStateMachineTable()
```

```
>+> if (stateMachineTable == null) makeStateMachineTable()
```

```
> }
```

注意调用 `StateMachineFactory.make()` 时的参数 `this`, 就是前面所创建的具体 `LocalizedResource` 的对象, 在这里变成了 `this.operand`, 即 `InternalStateMachine.operand`。也就是说, 作为一个具体对象的状态机, 其内部成分 `operand` 就指向这个对象。这样, 为一具体 `LocalizedResource` 对象创建的状态机, 里面就有着通向这个 `LocalizedResource` 对象的渠道, 借此可以访问这个对象的相关成分。

至此, 这个 `LocalizedResource` 对象已经建立了它自己的状态机。但是怎么才能让这个状态机动起来呢? 这里还有好多别的因素。为了深入理解它的状态机, 我们从另一个角度看一下 `LocalizedResource` 这个类的摘要。

```
class LocalizedResource implements EventHandler<ResourceEvent> {}
] LocalResourceRequest rsrc //具体本地资源请求, LocalizedResource 对象就是为此而建的
] Dispatcher dispatcher
] StateMachine<...> stateMachine
] Queue<ContainerId> ref; // Queue of containers using this localized resource
] static final StateMachineFactory<...> stateMachineFactory
] LocalizedResource(LocalResourceRequest rsrc, Dispatcher dispatcher)
] handle(ResourceEvent event)
    > ...
    > oldState = this.stateMachine.getCurrentState()
    > newState = this.stateMachine.doTransition(event.getType(), event)
] abstract class ResourceTransition implements SingleArcTransition<...> {}
] static class FetchResourceTransition extends ResourceTransition {}
]] transition(LocalizedResource rsrc, ResourceEvent event)
    > ResourceRequestEvent req = (ResourceRequestEvent) event
                                //参数 event 实际上是个 ResourceRequestEvent
    > LocalizerContext ctxt = req.getContext() //从中抽取 LocalizerContext
    > ContainerId container = ctxt.getContainerId()
                                //再从 LocalizerContext 中抽取 ContainerId
    > rsrc.ref.add(container) //将 ContainerId 加入 LocalizedResource 对象 rsrc
    > e = new LocalizerResourceRequestEvent(rsrc, ...,
                                req.getLocalResourceRequest().getPattern())
                                //创建(其实是重新构造)一个 LocalizerResourceRequestEvent 事件
    > rsrc.dispatcher.getEventHandler().handle(e) //并处理这个事件
] ...
] static class ReleaseTransition extends ResourceTransition {}
]] transition(LocalizedResource rsrc, ResourceEvent event)
```

`LocalizedResource` 类内部定义了一系列的内嵌类, 这里只列出了其中的两个, 就是 `FetchResourceTransition` 和 `ReleaseTransition`。实际上还有 `FetchSuccessTransition`、`FetchFailedTransition`、`LocalizedResourceTransition` 和 `RecoveredTransition`。之所以没有把它们都列出来, 是因为

这些类的结构都是一样的,除了构造方法之外就只有一个操作方法 `transition()`。不仅如此,连 `transition()` 的格局也都相同,里面免不了要创建一个事件,然后以此为参数调用 `rsrc.dispatcher.getEventHandler().handle()`,可谓千篇一律。其实,就此处的代码而言,与其说是创建一个事件,毋宁说是重构一个事件,因为这里用来创建 `LocalizerResourceRequestEvent` 事件的信息,是从参数 `event`,一个 `ResourceRequestEvent` 对象中抽取的,所以这里有着某种连贯性。

这里要说明一下,所谓一个“事件”,例如 `LocalizerResourceRequestEvent`,更贴切地说应该是“事件通知”或“事件报告”,而并非事件本身,但是大家都已习惯称之为“事件”。

细心的读者也许已经看出,前面作为参数 `hook` 出现在跳转规则中的那些用来提供伴随操作的,正是这里所定义的这些类的对象。比方说,如果当前状态是 `INIT`,而发生了事件 `REQUEST`,那么相应的操作就是调用由 `FetchResourceTransition` 提供的 `transition()` 方法。

除此之外,值得注意的是 `LocalizedResource` 还提供了一个方法 `handle()`,其调用参数是类型为 `ResourceEvent` 的事件。我们知道 `LocalizedResource` 实现了 `EventHandler` 界面,所以 `LocalizedResource.handle()` 也就是 `EventHandler.handle()`。不光是 `LocalizedResource` 如此,其他采用了状态机的类也是如此。事实上不仅是状态机,凡是实现了 `EventHandler` 界面的类,凡是代表着受事件驱动的过程的类,都有个 `handle()` 函数。再来看看上面 `FetchResourceTransition.transition()` 的代码中对 `rsrc.dispatcher.getEventHandler().handle(e)` 的调用,`rsrc.dispatcher` 是具体资源 `rsrc` 所绑定的 `Dispatcher`,而 `Dispatcher.getEventHandler()` 则确定应该把所产生的事件发送给哪一个 `EventHandler` 对象,这可以是另一个状态机,也可以是当前这个状态机本身,也可以不是状态机,但总之是代表着某个受事件驱动的过程。确定了之后,就调用那个对象的 `handle()` 函数,驱动那个过程前行。而那个过程的前行,则又可能反过来直接或间接发出驱动 `LocalizedResource` 的状态机进一步前行的事件。

事实上,`LocalizedResource` 的状态机就是靠这些因素的配合才能动起来的。下面我们就以这么一条跳变规则为例,来看看这个状态机是怎么运转的:

```
(ResourceState.INIT, ResourceState.DOWNLOADING,  
    ResourceEventType.REQUEST, new FetchResourceTransition())
```

假定现在这台状态机处于初始状态,即 `ResourceState.INIT` 状态。这时候发生了一个事件,有个对象发出了对于资源本地化的请求,于是就有了一个反映着这个事实的、类型为 `ResourceEventType.REQUEST` 的“事件”,成为状态机的一个触发条件。

那么这个事件究竟是怎么来的呢?这里介绍一下背景。它的来历是这样:在集群内的每个从节点上,`NodeManager` 会创建一个 `ContainerManagerImpl` 对象,顾名思义这是专门管理“容器(Container)”的,就好像是地方政府里的一个部门。这个对象又会创建另一个类型为 `ResourceLocalizationService` 的对象,这就好像是个资源本地化的专管员,它就专门负责一个具体容器的资源本地化。前面讲过,所谓容器其实就是把对于许多资源与任务的描述打包在一起。我们就从 `ResourceLocalizationService` 的这项操作 `handleInitContainerResources()` 开始,之所以会启动这个操作是因为得到了一个作为 `ContainerLocalizationRequestEvent` 的请求:

```
[ResourceLocalizationService.handleInitContainerResources()]
```

```

    /**
     * For each of the requested resources for a container, determines the
     * appropriate LocalResourcesTracker and forwards a LocalResourceRequest to that tracker.
     */
    private void handleInitContainerResources (ContainerLocalizationRequestEvent rsrcReqs) {
        Container c = rsrcReqs.getContainer(); //从容器本地化请求“事件”中取得具体的容器
        // create a loading cache for the file statuses
        LoadingCache<Path,Future<FileStatus>> statCache =
            CacheBuilder.newBuilder().build(FSDownload.createStatusCacheLoader(getConfig()));
        LocalizerContext ctxt =
            new LocalizerContext(c.getUser(), c.getContainerId(), ..., statCache); //创建一个上下文
        Map<LocalResourceVisibility, Collection<LocalResourceRequest>>
            rsrcs = rsrcReqs.getRequestedResources(); //获取所请求的资源清单
        for (Map.Entry<LocalResourceVisibility, Collection<...>> e : rsrcs.entrySet()) {
            //对于要求本地化的每种不同可见度的资源(资源集合中的每个二元组)
            LocalResourcesTracker tracker = getLocalResourcesTracker(e.getKey(), c.getUser(),
                c.getContainerId().getApplicationAttemptId().getApplicationId());
            //tracker 是个 EventHandler
            for (LocalResourceRequest req : e.getValue()) { //对于其中的每项资源
                tracker.handle(new ResourceRequestEvent(req, e.getKey(), ctxt));
                //调用这个 EventHandler 的 handle()函数
            }
        }
    }
}

```

如代码前面的注释所言,这个函数的作用是:就容器中所要求的每一项资源,确定其相应的 LocalResourcesTracker,并向其发送一个 LocalResourceRequest 事件。

从代码中看,容器中描述的资源是成组的,每个组是一个 entrySet,里面可以包含多个本地资源请求,所以有两层 for 循环嵌套。每一组资源都有个 LocalResourcesTracker,即本地资源的“追踪者”。具体的 LocalResourcesTracker 是由三个因素决定的:第一个是“可见度” LocalResourceVisibility,具体有 PUBLIC、PRIVATE 和 APPLICATION 三种;第二个是任务所属的用户;第三个是 ApplicationId。对于同一个“追踪者”可以发送多个具体的资源请求。

针对每一项具体的资源请求,都要发送一个事件,但是这个事件并非作为参数传下来的 ContainerLocalizationRequestEvent,而要根据从中抽取的每项资源信息另行创建一个 ResourceRequestEvent 类的事件,这个类的构造函数如下:

```

class ResourceRequestEvent extends ResourceEvent {
    ResourceRequestEvent(LocalResourceRequest resource,
        LocalResourceVisibility vis, LocalizerContext context) {
        super(resource, ResourceEventType.REQUEST); //这就是对于状态机的一个触发条件
        this.vis = vis;
    }
}

```



```

        this.context = context;
    }
    ...
}

```

这个类是对 ResourceEvent 的扩充,具体的资源和事件的类型都需要保存在 ResourceEvent 中,所以这里通过 super() 语句调用父类 ResourceEvent 的构造函数。我们此刻所关心的是,这个事件的类型是 ResourceEventType.REQUEST,这正是前面那条跳变规则的触发条件。

创建一个 ResourceEvent 类(更确切地说是 ResourceEvent 的某个子类)的事件,并通过某个 LocalResourcesTracker 的 handle() 函数加以处理,可以看作是资源本地化状态机的一个活动周期的起源。

这个事件,即请求,首先交给 tracker 的 handle(),由其加以处理。不管具体的 tracker 是谁,都是实现了 LocalResourcesTracker 界面的某类对象,实际上是 LocalResourcesTrackerImpl。界面 LocalResourcesTracker 是对 EventHandler 的扩充,所以 LocalResourcesTrackerImpl 同时也是个 EventHandler。LocalResourcesTrackerImpl.handle() 的代码中有一些我们此刻并不关心的内容,所以只看一下它与 REQUEST 事件相关的摘要:

```

[ResourceLocalizationService.handleInitContainerResources()
> LocalResourcesTrackerImpl.handle()]

```

LocalResourcesTrackerImpl.handle(ResourceEvent event)

```

> LocalResourceRequest req = event.getLocalResourceRequest()
> LocalizedResource rsrc = localrsrc.get(req) //rsrc 是个 LocalizedResource 对象
//localrsrc 是个 MAP,记载着已知的 LocalizedResource
> switch (event.getType()) {
> case REQUEST:
>+ if (null == rsrc) { //如果 MAP 中还没有记载此项资源
>++ rsrc = new LocalizedResource(req, dispatcher);
>++ localrsrc.put(req, rsrc); //就创建一项记载
>+ }
> }
> rsrc.handle(event) //所以实际上是 LocalizedResource.handle()

```

可见,LocalResourcesTrackerImpl 对象(对于事件)的处理,其实是由 LocalizedResource 对象进行的,而 LocalizedResource 正是我们现在所关心的那个状态机的拥有者。这样,我们就明白触发这个状态机的事件是怎么来的了。我们在前面已经有了 LocalizedResource 的摘要,现在再着重看它的 handle() 方法究竟干了些什么,这次是源代码:

```

[ResourceLocalizationService.handleInitContainerResources()
> LocalResourcesTrackerImpl.handle() > LocalizedResource.handle()]

```

```

LocalizedResource.handle(ResourceEvent event) {
    try {
        this.writeLock.lock(); //加锁
        Path resourcePath = event.getLocalResourceRequest().getPath(); //文件路径
        LOG.debug("Processing " + resourcePath + " of type " + event.getType());

        ResourceState oldState = this.stateMachine.getCurrentState(); //获取状态机的当前状态
        ResourceState newState = null;
        try {
            newState = this.stateMachine.doTransition(event.getType(), event);
        } catch (InvalidStateTransitionException e) {
            LOG.warn("Can't handle this event at current state", e);
        }
        if (oldState != newState) { //如果发生状态变化就记入日志
            LOG.info("Resource " + resourcePath + (localPath != null ?
                "(->" + localPath + "): " + " transitioned from " + oldState + " to " + newState);
        }
    } finally {
        this.writeLock.unlock(); //解锁
    }
}

```

这段程序,一言以蔽之,就是在加锁防干扰的条件下调用状态机的 doTransition() 方法。别的语句都只是为 LOG 即运行日志服务。不过从这些语句中可以看出,当程序从 doTransition() 返回的时候,状态机的跳变已经完成了,但是也可能前后状态相同。

那么状态机的 doTransition() 方法究竟干些什么呢? LocalizedResource 类的代码中的 stateMachine,其类型名为 StateMachine,但是 StateMachine 只是界面,实际的状态机是定义于 StateMachineFactory 内部的 InternalStateMachine。所以 stateMachine.doTransition() 实际上是 StateMachineFactory.InternalStateMachine.doTransition()。

```

[ResourceLocalizationService.handleInitContainerResources()
> LocalResourcesTrackerImpl.handle() > LocalizedResource.handle()
> StateMachineFactory.InternalStateMachine.doTransition()]

```

```

public synchronized STATE doTransition(EVENTTYPE eventType, EVENT event) {
    currentState = StateMachineFactory.this.doTransition(operand,
        currentState, eventType, event);
    return currentState;
}

```

只是转了一下手,把事情交给了 StateMachineFactory.this.doTransition(), 实际上就是 StateMachineFactory.doTransition()。如前所述,这里两个调用参数的类型都是抽象类型,但

是具体到 LocalizedResource,都已落实为 ResourceEventType 和 ResourceEvent。

但是要注意,状态机的当前状态,即 currentState,却正是在这里改变的。这里作为参数传递下去的 currentState 是老的当前状态,而返回的却是新的当前状态。所以,StateMachineFactory.doTransition()的作用一方面是执行伴随着本次状态跳变的操作,另一方面是返回新的状态,完成本次跳变。我们看一下这个函数的摘要:

```
[ResourceLocalizationService.handleInitContainerResources()
> LocalResourcesTrackerImpl.handle() > LocalizedResource.handle()
> StateMachineFactory.doTransition()]

doTransition(OPERAND operand, STATE oldState, EVENTTYPE eventType, EVENT event)
> transitionMap = stateMachineTable.get(oldState) //找到与当前状态相应的 transitionMap
> if (transitionMap!= null) {
>+ Transition<...> transition = transitionMap.get(eventType)
//根据事件类型在此 transitionMap 中找到相应的跳变说明
>+ if (transition!= null) return transition.doTransition(operand, oldState, event, eventType)
== StateMachineFactory.SingleInternalArc.doTransition() //执行规定的跳变
>+> if (hook!= null){
>+>+ hook.transition(operand, event) //执行伴随操作并完成状态变化
>+> }
>+> return postState; //返回跳变后的状态,即本次跳变的目标
> }
```

这里 4 个调用参数的类型都是来自模版的抽象类型,对于 LocalizedResource 来说 operand 的类型是 LocalizedResource;oldState 和 eventType 的类型分别是 ResourceState 和 ResourceEventType;event 的类型则是 ResourceEvent。

StateMachineFactory.doTransition()的程序也很简单,就是先根据当前状态 oldState 从状态跳转表中获取与此对应的 transitionMap,再从中找到以 eventType 为触发条件的那条规则,就是程序中的 transition。找到了适用规则,就调用其 doTransition()。我们在前面已经看到,这个 transition 的类型 Transition 是个界面,实际上是定义于 StateMachineFactory 内部,实现了这个界面的 SingleInternalArc,所以 Transition.doTransition() 实际上是 SingleInternalArc.doTransition(),而后者只是通过其 hook 调用 hook.transition()。当然,这是某个具体的 SingleInternalArc。

那么现在这个 hook 究竟是什么呢?我们在前面创建状态机工厂的过程中已经看到,那就是调用 addTransition()添加相应跳变规则时的参数 hook,在我们现在这个情景中是 FetchResourceTransition,因为当初那次调用是这样的:

```
addTransition(ResourceState.INIT, ResourceState.DOWNLOADING,
ResourceEventType.REQUEST, new FetchResourceTransition())
```

如前所述,FetchResourceTransition 是对抽象类 ResourceTransition 的扩充。后者虽然在形式上“实现”了界面 SingleArcTransition,实际上却要靠扩充落实了这个抽象类的具体类

来提供这个界面所定义的方法 `transition()`, 所以, 如 `FetchResourceTransition` 等这些具体的类都提供了自己的 `transition()`。

这样, 对 `hook.transition()` 的调用就转化为对 `FetchResourceTransition.transition()` 的调用。

```
[ResourceLocalizationService.handleInitContainerResources() >
LocalResourcesTrackerImpl.handle() > LocalizedResource.handle() >
StateMachineFactory.doTransition() > StateMachineFactory.SingleInternalArc.doTransition() >
FetchResourceTransition.transition()]
```

```
private static class FetchResourceTransition extends ResourceTransition {
    @Override
    public void transition(LocalizedResource rsrc, ResourceEvent event) {
        ResourceRequestEvent req = (ResourceRequestEvent) event; //还原成扩展后的类型
        LocalizerContext ctxt = req.getContext(); //从资源请求事件中获取 Context
        ContainerId container = ctxt.getContainerId(); //从 Context 中获取 ContainerId
        rsrc.ref.add(container); //将此 ContainerId 加入该项资源的引用链表
        rsrc.dispatcher.getEventHandler().handle(
            new LocalizerResourceRequestEvent(rsrc, req.getVisibility(), ctxt,
                req.getLocalResourceRequest().getPattern()));
    }
}
```

这里的参数 `rsrc` 是个 `LocalizedResource` 对象, 这就是前面调用 `doTransition()` 时的参数的 operand, 就是具体状态机的拥有者, 那就是某项具体的资源。

程序中先将作为参数传下来的 `event` 还原成其实际的、扩充后的类型。之所以需要有这样的还原, 是因为: 以 `LocalizedResource` 的状态机为例, 出于通用的考虑, 前面的代码中都不以像 `ResourceRequestEvent` 这样经过扩充后的类型作为参数的类型, 因为如果那样就得还有以 `ResourceLocalizedEvent`、`ResourceFailedLocalizationEvent` 等为参数类型的相同的代码。可是既然这些类型都是对 `ResourceEvent` 的扩充, 就不妨以 `ResourceEvent` 为参数类型, 这样一路下来的代码就可以通用了。不过, 虽然参数 `event` 是作为 `ResourceEvent` 类对象传下来的, 这并不意味着它真的就变成了 `ResourceEvent`, 也不能只把它当成 `ResourceEvent`, 那样就会丢失一部分信息。它的头部固然是 `ResourceEvent`, 但是尾巴也并没有被切掉。但是一旦到达了知道其底细的地方, 就要把它投射 (cast) 恢复成原来的类型, 因为它们的尾部还携带着信息。

然后, 作为 `ResourceRequestEvent`, 就可以从中恢复出相关的信息了, 在这里是容器的 `ContainerId`; 并进行相关的处理, 在这里只是把 `ContainerId` 加入到一个链表中, 把它保存下来。

最后一步的操作几乎是标准的, 那就是先另外创建一个事件, 在这里是 `LocalizerResourceRequestEvent`, 然后通过 `Dispatcher` 来触发别的受事件驱动的过程, 就是这里的 `rsrc.dispatcher.getEventHandler().handle()`, 这在前面已经讲到过了。

就像机器总是有固定和活动两种部件一样,我们也可以从概念上将状态机分成静态和动态两部分。静态部分就是状态和跳变规则,那一部分可以画成一个状态图;动态部分则是怎么使状态机转动起来的那套机制。现在,状态机已经动了一下,发生了一次状态跳变,但是还有没有下一次转动呢?状态机的转动是否可持续呢?Dispatcher 就起着十分重要的作用。

注意,Dispatcher 和 EventHandler 都是界面,实现了 Dispatcher 界面的主要是 AsyncDispatcher,摘要如下:

```
class AsyncDispatcher extends AbstractService implements Dispatcher {}
] BlockingQueue<Event> eventQueue           //事件队列
] EventHandler handlerInstance               //事件处理器
] Thread eventHandlingThread                //事件处理线程
] Map<Class<?extends Enum>, EventHandler> eventDispatchers
] AsyncDispatcher(BlockingQueue<Event> eventQueue)
    > super("Dispatcher")
    > this.eventQueue = eventQueue
    > this.eventDispatchers = new HashMap<Class<?extends Enum>, EventHandler>()
] serviceStart()
    > super.serviceStart()
    > eventHandlingThread = new Thread(createThread()) //创建事件处理线程
    > eventHandlingThread.start()
] createThread() //事件处理线程的代码
    > return new Runnable()
        ] run()
            > while (!stopped && ! Thread.currentThread().isInterrupted()) {
                >+ Event event = eventQueue.take() //从事件队列中取下一个事件
                >+ if (event != null) dispatch(event) //发起处理
            > }
] dispatch(Event event) //发起处理一个事件
    > Class<?extends Enum> type = event.getType().getDeclaringClass()
    > EventHandler handler = eventDispatchers.get(type)
    > if(handler != null) handler.handle(event) //调用事件处理器的 handle()函数
] register(Class<?extends Enum> eventType, EventHandler handler)
] getEventHandler() //获取本 Dispatcher 的事件处理器
    > if (handlerInstance == null) {
        >+ handlerInstance = new GenericEventHandler()
    > }
    > return handlerInstance
] class GenericEventHandler implements EventHandler<Event> {}
]] handle(Event event)
```

AsyncDispatcher 之所以称为“异步”,就是因为它有一个事件队列和一个事件处理线程。

这样,当有事件发生时,如果状态机尚未完成上一事件的处理,就不必睡眠等待,只要把事件挂入队列就完成并返回,事件处理线程执行完对先前事件的处理后自会来此队列中逐个加以处理。

注意这里的 `getEventHandler()` 返回的是一个 `GenericEventHandler` 对象,由此可见 `AsyncDispatcher.getEventHandler().handle()` 显然就是 `GenericEventHandler.handle()`。这是定义于 `AsyncDispatcher` 内部的一个类。所以,前面的 `rsrc.dispatcher.getEventHandler().handle()` 实际上就是 `AsyncDispatcher.GenericEventHandler.handle()`。调用这个函数本来的目的是要对事件进行处理,但实际上只是把待处理的事件挂入了 `AsyncDispatcher` 的事件队列,以等待处理。

在深入了解 `GenericEventHandler.handle()` 之前,我们还要先看一下上面所创建的事件 `LocalizerResourceRequestEvent()` 及其构造函数。

```
public class LocalizerResourceRequestEvent extends LocalizerEvent {
    private final LocalizerContext context;           //上下文
    private final LocalizedResource resource;         //所属的那项资源
    private final LocalResourceVisibility vis;
    private final String pattern;

    public LocalizerResourceRequestEvent(LocalizedResource resource,
                                         LocalResourceVisibility vis, LocalizerContext context, String pattern) {
        super(LocalizerEventType.REQUEST_RESOURCE_LOCALIZATION,
              ConverterUtils.toString(context.getContainerId()));
        this.vis = vis;
        this.context = context;
        this.resource = resource;
        this.pattern = pattern;
    }
    ...
}
```

这个类是对 `LocalizerEvent` 的扩充,这里通过 `super()` 调用其父类 `LocalizerEvent` 的构造函数,将其事件类型设置成 `REQUEST_RESOURCE_LOCALIZATION`,并设置其 `ContainerId`。

创建了这个事件之后,就以此为参数调用 `GenericEventHandler.handle()`。

```
[ResourceLocalizationService.handleInitContainerResources() >
LocalResourcesTrackerImpl.handle() > LocalizedResource.handle() >
StateMachineFactory.doTransition() > StateMachineFactory.SingleInternalArc.doTransition() >
FetchResourceTransition.transition() > AsyncDispatcher.GenericEventHandler.handle()]
```

```
public void handle(Event event) {
    if (blockNewEvents) return;
```

```

drained = false;
/* all this method does is enqueue all the events onto the queue */
int qSize = eventQueue.size();
if (qSize != 0 && qSize % 1000 == 0) {
    LOG.info("Size of event - queue is " + qSize); //每隔 1000 个事件做一次 LOG
}
int remCapacity = eventQueue.remainingCapacity();
if (remCapacity < 1000) { //队列的剩余容量已小于 1000, LOG 告警
    LOG.warn("Very low remaining capacity in the event - queue: " + remCapacity);
}
try {
    eventQueue.put(event); //将事件挂入队列
} catch (InterruptedException e) {
    if (!stopped) {
        LOG.warn("AsyncDispatcher thread interrupted", e);
    }
    throw new YarnRuntimeException(e);
}
}

```

显然,这里所谓的 `handle()`,所谓对于事件的处理,只是把它挂入了 `AsyncDispatcher` 的事件队列 `eventQueue`。把事件挂入队列之后,当前的这个流程就逐层返回了,这里包含了从 `hook.transition()` 和 `StateMachineFactory.doTransition()` 的返回,一直要返回到前面的 `ResourceLocalizationService.handleInitContainerResources()` 中产生 `ResourceRequestEvent` 事件并着手处理的地方,从那里对 `tracker.handle()` 的调用返回。这里特别要说明,从 `hook.transition()` 返回则意味着跳变伴随操作的完成,从 `StateMachineFactory.doTransition()` 返回意味着跳变业已完成,状态机已从 `INIT` 状态跳变到 `DOWNLOADING` 状态。

这就是异步处理的方式,把事件挂入一个队列,交给别人去处理就返回了,而不是事必躬亲地一直把事情处理完才返回,那样就是“同步”的了。

每个具体的 `AsyncDispatcher` 对象,在其创建之初,在 `serviceStart()` 中会创建一个线程 `eventHandlingThread`,这个线程所执行的程序是由 `createThread()` 创建的一个无名 `Runnable`。这个 `Runnable` 的主体 `run()` 的代码是一个 `while` 循环。只要不被打断,这个 `while` 循环实际上就是无穷的,每次循环都试图从事件队列 `eventQueue` 中获取一个事件(如果没有就睡眠等待),拿到一个事件就 `dispatch()` 这个事件。而 `dispatch()`,则要根据这个事件的类型从一个对照表 `eventDispatchers` 中寻找适合此种事件类型的事件处理器 `EventHandler`,若能找到就调用其 `handle()` 函数做进一步的处理。这就是“分发”,即 `dispatch`。

```

AsyncDispatcher.eventHandlingThread.run()
> while (!stopped && !Thread.currentThread().isInterrupted()) {
>+ event = eventQueue.take()
>+ dispatch(event)

```

```

>+> type = event.getType().getDeclaringClass()
>+> EventHandler handler = eventDispatchers.get(type)
>+> handler.handle(event)
> }

```

那么对照表即 eventDispatchers 中的内容是怎么来的呢? AsyncDispatcher 类提供了一个函数 register(), 具体的 EventHandler 可以调用这个函数向某个 AsyncDispatcher 对象登记, 这样就进入了它的对照表 eventDispatchers。

LocalizerResourceRequestEvent 是对 LocalizerEvent 的扩充, 而 LocalizerEvent 的处理者是 LocalizerTracker, 这是在 ResourceLocalizationService 的 serviceInit() 中通过一个语句 dispatcher.register(LocalizerEventType.class, localizerTracker) 登记的。所以, 这里的 handler.handle() 就是 LocalizerTracker.handle():

```

[AsyncDispatcher.eventHandlingThread.run() > AsyncDispatcher.dispatch() >
LocalizerTracker.handle()]

```

```

public void handle(LocalizerEvent event) {
    String locId = event.getLocalizerId();
    switch (event.getType()) {
    case REQUEST_RESOURCE_LOCALIZATION:
        // 0) find running localizer or start new thread
        LocalizerResourceRequestEvent req = (LocalizerResourceRequestEvent)event;
        // 将被当成 LocalizerEvent 的 event 还原成 LocalizerResourceRequestEvent
        switch (req.getVisibility()) {
        case PUBLIC:
            publicLocalizer.addResource(req);
            break;
        case PRIVATE:
        case APPLICATION:
            synchronized (privLocalizers) {
                LocalizerRunner localizer = privLocalizers.get(locId);
                if (null == localizer) {
                    LOG.info("Created localizer for " + locId);
                    localizer = new LocalizerRunner(req.getContext(), locId);
                    // 这是一个线程, 负责实施资源本地化
                    privLocalizers.put(locId, localizer);
                    localizer.start();
                }
            }
            // 1) propagate event
            localizer.addResource(req);
        }
    }
}

```

```
        break;
    }
    break;
}
```

这样,该项资源的本地化过程就又向前推进了,作为其一部分的 LocalizedResource 的这个状态机也向前走了一步,进入了 DOWNLOADING 状态。

资源本地化只是我们用来解释说明 Hadoop 代码中的状态机所用的例子,我们此刻关心的是状态机这种机制的实现,而不是资源本地化这个过程本身,所以到了这里我们就可以打住了。因为这下面肯定会有一些具体的操作,然后又向这个 LocalizedResource 对象发出下一个事件,于是这 LocalizedResource 对象的状态机就又会再往前走一步。

在涉及状态机的整个“生态系统”中,通常有三个线程即三个主体在活动。一是状态机的所有者,也是状态机的操作者,是它因某种请求或事件的到来而调用 handle()函数,引起了状态机的跳转以及伴随着跳转的响应,其结果是产生了另一个事件;二是分发者 Dispatcher,它将状态机在响应操作中产生的事件分发给某个与状态机所代表的过程相关的对象,触发其某些方面的操作;三就是这个与状态机所代表过程相关的对象,例如上面的 LocalizerRunner,在 Dispatcher 所转达的事件触发下进行某些处理,然后又向状态机的操作者发出新的请求或事件。如此周而复始,直至该状态机所实现的整个过程结束。

其中 Dispatcher 所起的作用类似于网络中的路由器,它只是起着路由的作用,所以既可以用于状态机,也可以用于别处,当然作为分发目标的对象必须实现 EventHandler 界面,即提供 handle(T event)函数。

LocalizedResource 对象的状态机是这样,其他对象的状态机(如果有的话)也是这样。

总结一下,假定 A 是一个类,a 是这个类的一个对象,并且 A 有状态机,则有:

(1)StateMachineFactory 是 A 的静态成分,所以一共只有一份,在 A.class 中,而具体的 A 类对象中并不包含 StateMachineFactory。

(2)每个 A 类对象都会有由此 StateMachineFactory 生成的状态机 stateMachine,状态机的主体是一个跳转表,表中是一些跳转规则。可以把每条跳转规则看成一个四元组(当前状态,新状态,触发事件,hook),其中的 hook 是一个用来对状态跳变做出伴随操作的对象,每个 hook 对象都提供一个 transition()函数。不过,规则中也可以没有 hook,那就说明除状态跳变外无须任何反应。

(3)除状态机外,还需要有个实现了 Dispatcher 界面的某类对象,通常是 AsyncDispatcher 对象。

(4)发生某种与此状态机有关的事件时,会产生一个定义于这个 StateMachine 的事件对象,并以此事件对象为参数调用 A.handle()。

(5)A.handle()根据状态机 A.stateMachine 的当前状态和具体事件对象的类型在其跳转表中找到相应的跳转规则 T。

(6)如果本条跳转规则中提供了一个用来对该次跳转做出伴随操作的对象 hook,例如 FetchResourceTransition,那就调用 hook.transition()。注意跳转前后的状态也可以相同,即

实际上没有状态变化。

(7)然后将状态机的当前状态改变成跳转规则所规定的新状态。

(8)在 `hook.transition()` 中很可能会创建新的事件对象(例如前面的 `LocalizerResourceRequestEvent`)，这个事件对象要通过预设的 `Dispatcher` 对象加以发送，最终调用某个目标对象的 `handle()` 函数。这个分发目标可以就是 A 本身，也可以是别的对象，需要预先向 `Dispatcher` 对象登记。

这样，一个拥有状态机的对象 A，从它的 `handle()` 第一次被调用开始，每次受到事件触发时根据其当前状态和触发事件类型确定调用某个 `hook` 对象的 `transition()` 以做出行动上的反应，并且改变或不改变其状态。而 `hook` 对象在作为反应的行动中又可能产生另一个事件，并通过 `Dispatcher` 加以发送。最后，直接或间接地又会以新的事件对象为参数调用对象 A 的 `handle()`，它的状态机就是这样动了起来，运转了起来。

Hadoop 中所有的状态机都是这样，都按照这个标准的模式运转。以后，当涉及状态机时，我们就不再深入到这样的细节中去，而只需简单说一下谁的状态机在这个特定事件的触发下做出何种反应、跳变到何种状态；如果创建新的事件，则这个新的事件被分发到何处、触发何种行为，这样就行了。

3.5 资源管理器 ResourceManager

如前所述，在一个 Hadoop 集群中，有且只有一个实际有效的“资源管理者”，即 `ResourceManager` 类的对象，起着 Hadoop 平台核心，其实是 YARN 框架核心的作用，管理着整个平台（集群）的计算资源，可以说相当于“中央政府”。对于宿主机而言，执行着 `ResourceManager` 的 Java 虚拟机是个独立的进程。

除 `ResourceManager` (缩写为 RM) 这个“主节点”以外，集群中凡启动了 YARN 子系统的所有节点都是 `NodeManager` (缩写为 NM) 节点，都是“从节点”，唯一例外的是后面会讲到的备份 RM，或者说不活跃的 RM。RM 与 NM 之间是主从关系，相当于中央与地方的关系。注意，这里说的是“主从”而不是“主次”，更不是主副，因为主次之间不一定是从属的关系、领导与被领导的关系。

用户每向这个 Hadoop 平台提交一个应用程序作业，即 App，资源管理器就会设法在某个 NM 节点上为其另起一个 Java 虚拟机以运行 App 管理者 (AppMaster, AM)。YARN 的 AM 不止一种，什么样的 App 就用什么样的 AM。对于 MapReduce 应用，那就是 `MRAppMaster`。`MRAppMaster` 在独立的 Java 虚拟机上运行，与当地的 `NodeManager` 不在同一个 Java 虚拟机上。

可想而知，`ResourceManager` 类的体积一定不小，我们得要分层次做它的摘要。当然，既然是摘要就不会是全面的。

```
class ResourceManager extends CompositeService implements Recoverable {}
] List<Service> serviceList = new ArrayList<Service>() //这是在 CompositeService 中
] Dispatcher rmDispatcher //这是一个 AsyncDispatcher
] AdminService adminService
] RMActiveServices activeServices //见后
```



```

] ResourceScheduler scheduler           //资源调度器,有三种选择
] ClientRMService clientRM
] ApplicationMasterService masterService //为现有的 AM 提供服务和管理
] NodesListManager nodesListManager
] RMAppManager rmAppManager             //管理已提交而尚未完成的 App
] WebApp webApp                         //提供作为 webApp 的网站服务
] AppReportFetcher fetcher
] ResourceTrackerService resourceTracker
] String webAppAddress
] main(String argv[])
    > resourceManager = new ResourceManager()
    > resourceManager.init(conf) //这会调用 super.init(),然后回过来调用这里的 serviceInit()
    > resourceManager.start()
] serviceInit(Configuration conf)
    > doSecureLogin() //如果启用了安全机制,用户就得登录,此后使用实际用户的身份
    > rmDispatcher = setupDispatcher() //创建一个 Dispatcher 并将其设置成 RM 的 Dispatcher
    > adminService = createAdminService() //创建系统管理服务
    > createAndInitActiveServices() //创建许多的 service 并放进上面的 serviceList 中
    > super.serviceInit(this.conf)
] serviceStart()
    > rmStore.start()
    > super.serviceStart()
] class RMActiveServices
] class ApplicationEventDispatcher implements EventHandler<RMAppEvent> {}
] class RMContainerPreemptEventDispatcher{ }
] class ApplicationAttemptEventDispatcher implements EventHandler<...> {}
] class NodeEventDispatcher implements EventHandler<RMNodeEvent> {}
] class SchedulerEventDispatcher{ }
]] class EventProcessor implements Runnable {}
]] handle(SchedulerEvent event)

```

在这份摘要里,除 webAppAddress 以外,在 main()之前的都是 ResourceManager 内部第一层的结构成分,相当于中央一级的部门,其中有些是特别重要的,像 rmDispatcher、scheduler 等都是。

ResourceManager 对象带有 main()函数,说明一般而言这个类的对象都是独占一台 JVM 的,同一台 JVM 上的其余对象都由它直接或间接创建。注意,像这一类的对象是 JVM 装载并调用其 main()在先,此时尚未创建具体的对象,对象的创建一般是由 main()完成的。

RM 对象的初始化由 serviceInit()完成。Hadoop 上的安全机制启用与否是可配置的,如果启用了安全机制,用户此时就得登录,登录以后便使用实际用户的身份;否则就直接使用启动该 JVM 的用户,通常是特权用户的身份了。RM 并没有状态机,但也需要有个通用的 Dispatcher,因为在运行过程中它也会产生事件而需要分发。当然,想要采用这个通用

Dispatcher 就得向其登记,告知其什么样的事件应该发送给谁。细看一下 `setupDispatcher()` 的代码,就可发现 RM 所用的其实也是 `AsyncDispatcher`。下面的 `createAdminService()` 所创建的是个 `AdminService`,即管理服务对象,这与系统的容错机制也有关系,但是我们现在还不关心这个。

再下面的 `createAndInitActiveServices()` 就厉害了,它创建一个 `RMActiveServices` 类对象,在这个对象的初始化过程中为 RM 创建了许多“活跃”服务。前面讲过,一个集群中只有一个有效的,也即活跃的资源管理者,但是这并不意味着不可以有后备的、不活跃的 `ResourceManager`,二者的区别就在于活跃的 RM 上有着一些活跃服务,从而能提供并被允许提供这些服务。而 `RMActiveServices`,就是这些活跃服务的管理者。那么有些什么样的活跃服务呢?我们可以看一下 `RMActiveServices` 的摘要,特别是其 `serviceInit()` 方法的摘要:

```
class ResourceManager{}
] class RMActiveServices{}
]] EventHandler<SchedulerEvent> schedulerDispatcher
]] ApplicationMasterLauncher applicationMasterLauncher
]] serviceInit() //注意这是 RMActiveServices 的 serviceInit()
    > containerAllocationExpirer = new ContainerAllocationExpirer(rmDispatcher);
    > addService(containerAllocationExpirer);
    > rmContext.setContainerAllocationExpirer(containerAllocationExpirer)
    > //为减小篇幅,以下各项均略去 addService()和对 rmContext 的设置
    > amLivelinessMonitor = createAMLivelinessMonitor() //用来监视各 AM 是否存活
    >> new AMLivelinessMonitor(this.rmDispatcher)
    > amFinishingMonitor = createAMLivelinessMonitor() //用来监视各 AM 是否已完成使命
    > nlm = createNodeLabelManager() //创建 RMNodeLabelsManager
    > rmStore = RMStateStoreFactory.getStore(conf) //创建 RMStateStore,以保存各种副本
    > rmStore.setRMDispatcher(rmDispatcher)
    > rmApplicationHistoryWriter = createRMApplicationHistoryWriter() //用于运行日志 Log
    > systemMetricsPublisher = createSystemMetricsPublisher() //用来发布统计信息
    > nodesListManager = new NodesListManager(rmContext) //用来管理 NM 节点状态清单
    > scheduler = createScheduler() //创建资源调度器
    >> schedulerClazz = Class.forName(schedulerClassName) //从 conf 文件获取调度器类名
    >> ReflectionUtils.newInstance(schedulerClazz, this.conf) //创建调度器
    > schedulerDispatcher = createSchedulerEventDispatcher()
                                //创建调度器专用的 Dispatcher
    >> new SchedulerEventDispatcher(this.scheduler)
    > //创建三种 Dispatch 目标对象(接受来自 RM 的事件),并向 RM 的 Dispatcher 登记
    > rmDispatcher.register(RMAppEventType.class,
                           new ApplicationEventDispatcher(rmContext))
    > rmDispatcher.register(RMAppAttemptEventType.class,
                           new ApplicationAttemptEventDispatcher(rmContext))
    > rmDispatcher.register(RMNodeEventType.class,
```

```

        new NodeEventDispatcher(rmContext))
> nmLivelinessMonitor = createNMLivelinessMonitor() //用来监视 NodeManager 的存活
> resourceTracker = createResourceTrackerService() //用来跟踪资源的使用
> JvmMetrics.initSingleton("ResourceManager", null) //保证统计信息中只有一个 R
> reservationSystem = createReservationSystem() //如有要求就创建资源预订机制
>> reservationClazz = Class.forName(reservationClassName)
>> ReflectionUtils.newInstance(reservationClazz, this.conf)
> masterService = createApplicationMasterService() //用来为 NM 节点上的 AM 提供服务
>> new ApplicationMasterService(this.rmContext, scheduler)
> rmAppManager = createRMAppManager() //用来管理 AM
>> new RMAAppManager(this.rmContext, ...)
> clientRM = createClientRMService() //用来为客户提供资源服务
> applicationMasterLauncher = createAMLancher() //用来在 NM 节点上启动运行 AM
>> new ApplicationMasterLauncher(this.rmContext)
]] serviceStart()

```

可见 RM 上“活跃服务”的种类真是不少,用来提供这些服务的对象也真不少。创建了每一个这样的“服务”以后,都要通过 `addService()` 把它加入到 RM 的服务列表 `serviceList` 中,这里为了减小摘要的篇幅就把这些操作略去了。

这里需要说明一下 `Dispatcher`。RM 有个顶层的 `rmDispatcher`,用来帮助向 `ApplicationEventDispatcher`、`ApplicationAttemptEventDispatcher`、`NodeEventDispatcher` 等对象分发事件。但是这三者的命名容易令人误解,因为它们其实并非 `Dispatcher`,而是 `Dispatch` 的目标;它们实现的是 `EventHandler` 界面,而不是 `Dispatcher` 界面。这里在 `RMAActiveServices` 中又有个 `schedulerDispatcher`,这显然与调度器有关,其类型是 `SchedulerEventDispatcher`。但是它实现的也是 `EventHandler` 界面,而不是 `Dispatcher` 界面;从形式上看,它是用来处理 `SchedulerEvent` 类事件的,而不是帮助分发事件的路由器。

不过称它们为 `Dispatcher` 也并非全无道理,因为它们其实起着中介的作用,对于 `rmDispatcher` 来说,它们确实是事件的处理者而不是分发者,所实现的界面也是 `EventHandler` 而不是 `Dispatcher`。但是,对于站在它们身后的真正意义上的事件处理者而言,它们却相当于事件的分发者,相当于 `Dispatcher`。

这里的 `ApplicationEventDispatcher` 和 `ApplicationAttemptEventDispatcher`,前者有关 `Application`,用来驱动代表着具体 App 的 `ApplicationImpl` 对象中的状态机;后者有关 `ApplicationAttempt`,用来驱动代表着一次具体运行尝试的 `RMAAppAttemptImpl` 对象中的状态机。二者的区别在于:前者是宏观层面的,代表着一个 App;而后者则代表运行该 App 的一次具体尝试,相对而言是微观层面的。如果 App 的一次 `ApplicationAttempt` 失败,AM 可以要求再来一次,要到屡战屡败之后才能宣告 `Application` 的失败。

至于 `NodeEventDispatcher`,则与 `ResourceTrackerService` 和 `RMNodeImpl` 有关。RM 为 `NodeEventDispatcher` 登记要接收的事件类型是 `RMNodeEventType`,这种事件主要来自 `ResourceTrackerService`,反映着集群中各节点的状态变化,例如失去连接然后又恢复了,或者机器重启了,等等。`NodeEventDispatcher` 用这些事件驱动相应的 `RMNodeImpl` 对象中与此

节点相对应的状态机。NodeEventDispatcher 的 handle() 函数说明了这一点:

```
[ResourceManager.NodeEventDispatcher.handle()]
```

```
public void handle(RMNodeEvent event) {
    NodeId nodeId = event.getNodeId();
    RMNode node = this.rmContext.getRMNodes().get(nodeId);
    //RMNode 是界面, RMNodeImpl 实现了 RMNode 和 EventHandler 两个界面
    if (node != null) {
        try {
            ((EventHandler<RMNodeEvent>) node).handle(event);
            //操作 RMNodeImpl 的状态机, 调用其 doTransition()
        } catch (Throwable t) {
            LOG.error("Error in handling event type " + event.getType() + " for node " + nodeId, t);
        }
    }
}
```

对于集群中的每一个节点, RM 维持着一个与之对应的 RMNodeImpl 对象, 在这个对象中有一个状态机, 代表着该节点的当前状态。当然, 这样的 RMNodeImpl 对象有很多, 而 NodeEventDispatcher 就是这些状态机的驱动者。它根据到来的事件 event 内容中的 NodeId 确定这是要分发到哪一个节点的状态机, 然后找到代表着那个节点的 RMNode 对象 node, 即 RMNodeImpl, 就以 event 为参数直接调用其 handle() 函数。从这个意义上说, 称之为 Dispatcher 确实也有道理, 并且这是个同步的 Dispatcher。

回到前面对 ResourceManager 数据部分的摘要, 其实还有更多, 下面先择要简单介绍几个, 其余的会在后面结合具体的情景加以介绍, 这里就不一一列举介绍了。

3.6 资源调度器 ResourceScheduler

RM 中最重要的活跃服务提供者应该是“资源调度器(scheduler)”, 这是个实现了 ResourceScheduler 界面的某类对象, 通过 createScheduler() 创建, 具体的类型则在配置文件中加以指定。注意这里所说的只是资源调度, 实际上也是对于“作业(Job)”投运的调度, 这跟操作系统中进程/线程的运行调度是两码事。Hadoop 提供了三种不同的资源调度器供选用, 用户可以在配置文件中加以选择。这三种调度器是:

(1)FifoScheduler。这是最简单的调度器, 其调度策略就是“先来先走”、“先来先给”, 按次序来。

(2)FairScheduler。这种调度器要考虑公平的问题, 不让属于同一用户的作业独占资源。

(3)CapacityScheduler。这是最复杂的调度器, 还要考虑尽量合理地使用各个节点的资源。

配置文件 yarn-site.xml 中有一条“yarn.resourcemanager.scheduler.class”, 就是用于这个目的。如果不加规定, 则默认的调度器 DEFAULT_RM_SCHEDULER 是 CapacityScheduler。

不过,从理解 Hadoop 的结构和运行流程的角度,这三者并无太大的不同,所以本书在涉及调度器时都以最简单的 `FifoScheduler` 为例,有兴趣或需要的读者可以自己分析一下另外两个调度器的代码。

调度当然不能是无米之炊,调度器手里的资源只能来自集群中的众多 `NodeManager` 节点。为此,`ResourceManager` 内部有个 `ResourceTrackerService` 类的对象 `resourceTracker`,它跟踪管理着 `NodeManager` 节点所知道的资源变动。另有一个 `NodesListManager` 类的对象 `nodesListManager` 则维持着一个节点清单,记录着哪些节点当前是可用的,哪些则是不可用的。而 `ResourceTrackerService` 和 `NodesListManager` 所掌握的信息,则来自众多 `NodeManager` 节点的心跳报告,即伴随着每次“心跳”提供的报告。这些信息,最终都要汇集到资源调度器,因为这些资源都要由资源调度器分配出去。

有了“家底”,调度器就可以应各个作业组长 `AM` 的要求通过 `allocate()` 分配资源、通过 `assignContainers()` 指派容器了。作业完成计算以后,还要通过 `completedContainer()` 回收容器。

对于 `ResourceManager` 内部的这些成分,以及其他一些成分,后面将结合具体的情景和流程加以介绍。

创建了前述的那么多活跃服务,完成了初始化之后,`RM`,即当前的 `Active RM`,便可坐等客户上门了。至于作为 `Standby` 的 `RM`,如果有的话,则不用创建这些活跃服务,只需要做好公共部分的初始化就行了;但是要做好准备,因为随时都可能需要接管 `Active RM` 的角色,那时候就要创建这些活跃服务了。

第 4 章

Hadoop 的 RPC 机制

4.1 RPC 与 RMI

RPC 是“Remote Procedure Call”即“远地过程调用”的缩写。这个机制的目的,是让一台机器上的程序能像调用本地的“过程”那样来调用别的机器上的某些过程。这里所谓“过程”,在传统的 C 程序设计中统称为“函数”,在 Pascal 程序设计中既可以是 PROCEDURE 也可以是 FUNCTION,在 Java 等 OO 程序设计语言中就是“方法(method)”。所以,Java 传统的 RPC 机制称为 RMI,即“远地方法启用(Remote Method Invocation)”。对 RPC 机制的要求是:从程序代码上看,过程的调用者就好像在调用本地函数一样,但是被调用过程的代码实际上在别的机器上,被调用的过程是在别的机器上执行,然后返回执行的结果,对调用者而言就像从本地的函数调用返回一样。在这个过程中,调用者(线程)发动调用之后就进入睡眠,直至调用返回时才被唤醒。显然,这种机制的实现,是以跨机器节点的进程间通信即 IPC 为基础的。所以,就调用者而言,虽然形式上就像本地的调用一样,但是花费的时间显然是不同的,而且调用失败的概率也会增加。因此,即使是调用最简单的、在本地的条件下绝不会失败的函数也得做好失败(Exception)的准备,把调用放在 try 程序段中,并安排好异常处理和对策。

需要特别说明的是,RPC 并非针对远地的所有过程,并不是对远地所有的过程都可以随心所欲地通过 RPC 加以调用,而只能针对预先确定的某些过程,并且在程序上得有些准备和安排。

Hadoop 本可以直接借用 Java 的 RMI 机制,但是出于种种考虑决定自搞一套 RPC 机制,这在很大程度上是因为有 Google 的开源软件 ProtocolBuffer 可资利用。而 Google 之所以要自己开发一套 ProtocolBuffer,应该是要以此为核心实现自己的 RPC 机制,主要应是出于软件开发效率、灵活性和多语种的考虑。

考虑到 RPC 往往涉及对象即数据结构的跨节点传输,这里面又有串行化(serialization)和去串行化的问题,所以,RPC 和 ProtocolBuffer 也并不简单。不过对于 Hadoop 而言这毕竟是很底层的东西,读者如果不想一开始就在这里陷得太深,也可以先跳过本章,到后面再回头来看 RPC 和 ProtocolBuffer 的细节。

如上所述,RPC 是建立在 IPC 的基础之上的,而现在最灵活、最通用的通信手段就是 Socket。

RPC 交互的两方,总有一方是通信的主动发起方,也是某种服务的需求方;另一方则是被

动的响应方,也是服务的提供方。所以,通信中至少有一方扮演着“服务者”即 Server 的角色。如果是双方对等的通信,则各自都有其作为 Server 的一面。在 Hadoop 的系统结构中,节点有主从(Master/Slave)之分,通常主节点扮演着 Server 的角色,主从节点间的通信都是由从节点发起的,主节点则像是“公仆”。而不同的节点之间的通信则是对等的,谁都可以发起,所以每个从节点也都有作为 Server 的一面。

为此,Hadoop 中定义了一个抽象类 Server,下面是它的摘要:

```
abstract class Server {} // org.apache.hadoop.ipc.Server
] Map<RPC.RpcKind, RpcKindMapValue> rpcKindMap
] Map<String, Class<?>> PROTOCOL_CACHE
] ThreadLocal<Server> SERVER = new ThreadLocal<Server>()
] ThreadLocal<Call> CurCall = new ThreadLocal<Call>()
] int handlerCount; // number of handler threads
] int readThreads; // number of read threads
] int readerPendingConnectionQueue; // number of connections to queue per read thread
] Class<? extends Writable> rpcRequestClass; // class used for deserializing the rpc request
] CallQueueManager<Call> callQueue
] ConnectionManager connectionManager
] Listener listener //listener 是个线程,其类型定义见下
] Responder responder
] Handler[] handlers
- - - - 以上是结构成分与数据部分,以下是对方法(函数)及内嵌类的定义部分 - - - -
] registerProtocolEngine(RPC.RpcKind rpcKind, ...) //登记一种 RpcKind 及其 ProtocolEngine
] getRpcInvoker(RPC.RpcKind rpcKind)
] class Call implements Schedulable {}
] class Listener extends Thread {}
]] Reader[] readers //Listener 对象内部的 Reader 线程数组
]] Listener() //Listener 的构造函数
> address = new InetSocketAddress(bindAddress, port) //Socket 地址和端口号
> // Create a new server socket and set to non blocking mode
> acceptChannel = ServerSocketChannel.open() //创建 Socket 通道
> acceptChannel.configureBlocking(false)
> bind(acceptChannel.socket(), address, backlogLength, conf, portRangeConfig) //绑定地址
> port = acceptChannel.socket().getLocalPort(); //Could be an ephemeral port
> selector = Selector.open() //create a selector,类似于 select()系统调用中的 fd_set
> readers = new Reader[readThreads] //创建一个线程数组
> for (int i = 0; i < readThreads; i++) { //预先创建一组 Reader 线程并加以启动
>+ Reader reader = new Reader("Socket Reader #" + (i + 1) + " for port " + port)
>+ readers[i] = reader
>+ reader.start() //start 所创建的 Reader 线程
> }
```

```

> // Register accepts on the server socket with the selector.
> acceptChannel.register(selector, SelectionKey.OP_ACCEPT)
> this.setName("IPC Server listener on " + port)
> this.setDaemon(true)    //断开该线程与标准 I/O 通道的连接,使其变成后台线程
]] run() //这是 Listener 线程的 run() 函数
]] class Reader extends Thread {} //接收线程 Reader,接受和处理服务请求
]]] Reader(String name)
    > super(name)
    > this.pendingConnections =
        new LinkedBlockingQueue<Connection>(readerPendingConnectionQueue)
    > this.readSelector = Selector.open()
]]] addConnection(Connection conn)
    > pendingConnections.put(conn)
    > readSelector.wakeup()
]] doRead(SelectionKey key)
    > Connection c = (Connection)key.attachment()
    > count = c.readAndProcess()
    > if (count < 0) closeConnection(c)
] class Responder extends Thread {} //回应线程 Responder
] class Connection {} //代表着一个具体的连接
]] getAuthorizedUgi(String authorizedId)
]] saslReadAndProcess(DataInputStream dis) //SASL 加密报文的读入和处理
]] saslProcess(RpcSaslProto saslMessage)
]] processSaslMessage(RpcSaslProto saslMessage) //处理加密的报文
]] ...
] class Handler extends Thread {} //Call 处理线程
] Server(String bindAddress, int port, Class<?extends Writable> rpcRequestClass,
    int handlerCount, int numReaders, int queueSizePerHandler, Configuration conf,
    String serverName, SecretManager<?extends TokenIdentifier> secretManager,
    String portRangeConfig) //Server 类对象的构造函数
    > this.bindAddress = bindAddress
    > this.conf = conf
    > this.portRangeConfig = portRangeConfig
    > this.port = port
    > ...
    > String prefix = getQueueClassPrefix()
    > this.callQueue = new CallQueueManager<Call>(getQueueClass(prefix, conf),
        maxQueueSize, prefix, conf)
    > this.secretManager = (SecretManager<TokenIdentifier>) secretManager
    > this.authorize = conf.getBoolean(

```

```

CommonConfigurationKeys.HADOOP_SECURITY_AUTHORIZATION, false)
> listener = new Listener() //创建一个 Listener 对象
> connectionManager = new ConnectionManager()
> responder = new Responder()
> if (secretManager != null || UserGroupInformation.isSecurityEnabled()) { //如果使用加密
>+ SaslRpcServer.init(conf)
>+ saslPropsResolver = SaslPropertiesResolver.getInstance(conf)
> }
> this.exceptionsHandler.addTerseExceptions(StandbyException.class)
] abstract Writable call(RPC.RpcKind rpcKind, String protocol,
                        Writable param, long receiveTime)

```

Server 是个抽象类, 未经扩充落实是不能为其创建具体对象的。为什么这是个抽象类, 是它的什么成分还一时无法落实呢? 其实只有一样, 那就是它的(带 4 个参数的)方法函数 call(), 这是个抽象方法, 因为在这里还不能落实, 无法为之提供具体的代码。然而这个 call() 是个十分重要的函数, 因为这就是 RPC 三个字母中的那个 C, 即 Call。RPC 的关键就是在远地调用某个函数, 但是具体怎么调用却与所用的“协议”即 protocol 有关。协议不同, 下面的代码就不同, 所以无法脱离具体的协议来提供实现这个方法的代码。显然, 能实际创建的 Server 至少得要补上这个 call() 方法的具体实现才行。在 Java 语言中有两种手段可以做到这个: 一种是静态定义一个扩充落实这个抽象类的实体类, 在里面提供 call() 方法的代码, 然后加以创建; 另一种是动态定义, 即在通过 new 语句创建 Server 对象时临时补上一个 call() 方法。

Hadoop 的代码中采用的办法是: 先采用静态定义的办法对 Server 加以扩充, 但暂不落实 call()。我们一般所说的子类扩充(extends)父类, 其实不仅仅是扩充, 而有着三方面的意思: 一是扩充, 就是增加数据成分和操作方法; 二是落实, 就是补上父类声称提供但实际并未实现的操作方法; 三是修改, 就是用同名同参数的操作方法覆盖替代父类所提供的操作方法。Hadoop 的代码中就是先搁下对 call() 的落实, 而先对抽象类 Server 进行扩充, 具体就是在另一个类 RPC 中定义了一个同名的内嵌抽象类 Server, 即 RPC.Server, 用来扩充落实抽象类 Server。但是这个 RPC.Server 仍是抽象类, 还需要进一步的扩充落实:

```
abstract static class Server extends org.apache.hadoop.ipc.Server {}
```

这是在 RPC 类内部, 所以实际定义的是 RPC.Server, 这是对 org.apache.hadoop.ipc.Server, 就是上面那个抽象类的扩充。但是, 这里声明了是 abstract, 所以仍需进一步对 RPC.Server 加以扩充落实, 才能为其创建具体对象。有意思的是, RPC.Server 其实已经补上了 call() 这个函数, 下面是它的代码摘要, 并已展开:

```

RPC.Server.call(RPC.RpcKind rpcKind, String protocol, Writable rpcRequest, long receiveTime)
> invoker = getRpcInvoker(rpcKind)
>> RpcKindMapValue val = rpcKindMap.get(rpcKind)
>> return (val == null)?null : val.rpcInvoker //这是从 getRpcInvoker() 返回
> return invoker.call(this, protocol, rpcRequest, receiveTime)

```

可是在 RPC.Server 的定义前面还是加上了 abstract 的标签。这样, Java 编译器就会把住

程序编译这一关,不让在程序中直接创建 `RPC.Server` 的对象,这当然是个故意的安排。这里的 `getRpcInvoker()` 是由 `org.apache.hadoop.ipc.Server` 这个类提供的,它根据给定的 `RPC` 类型 `rpcKind` 从一个 `rpcKindMap` 中获取其 `RpcInvoker`,然后调用这个 `RpcInvoker` 所提供的 `call()` 函数。问题是,如果 `getRpcInvoker()` 返回 `null` 呢? 如果没有措施保证 `rpcKindMap` 中确实有给定 `rpcKind` 的 `RpcInvoker`,那么返回 `null` 几乎是必然的。代码的设计者之所以要在 `RPC.Server` 的定义前面加上标签 `abstract`,就是为了防止这样的情况发生。至此为止我们还不能认为 `call()` 这个函数已经得到落实和解决。设计者的意图,是要另外用一个类来扩充 `RPC.Server`,在那里保证给定的 `rpcKind` 的 `RpcInvoker` 一定会在 `rpcKindMap` 中,而且那个 `RpcInvoker` 的 `call()` 一定是符合设计要求的 `call()`,然后为这个经过扩充的类创建对象。至于为什么不是一步到位,而要有这样一个中间步骤,应该是因为不同的 `rpcKind` 导致具体实现有较大的不同,所以本着尽量提取公因子的原则才会有这么中间的一步。

不过光是 `call()` 这个方法尚未落实解决并不妨碍我们对 `Server` 类其他方面工作机理的理解,再说我们现在暂时也并不需要知道对具体函数的调用。

简而言之,当创建一个 `Server` 对象时,如果不考虑加密等附加的细节,那么首先要在这个 `Server` 的构造函数中创建一个 `Listener` 对象,这是用来“倾听”连接请求的;同时还要创建一个 `Responder` 对象,这是用来向对方做出回应的;还有一个 `CallQueueManager`,用来管理 `Listener` 内部的请求队列。

注意,既然有 `Listener`,就说明底层的网络通信是有连接的,所以采用的是 `TCP` 而不是 `UDP` 协议。

当然,创建 `Listener` 和 `Responder` 这两个对象时要执行它们的构造函数。从摘要中可见,在 `Listener` 的构造函数中会创建 `Socket`,然后创建一个线程数组 `readers`,这个数组的大小为 `readThreads`,来自创建 `Server` 时的参数 `numReaders`。这表示 `Listener` 中需要有多少个用来响应连接请求的 `Reader` 线程,一个 `Server` 通常都需要维持多个并发连接,所以就需有多个 `Reader` 线程。为提高响应速度,这里预先创建了一组 `Reader` 线程并加以启动,而并非有连接建立时才临时创建线程。

创建并启动了这些线程之后,它们就各自进入了各自的 `run()` 函数。我们先看 `Listener` 的 `run()` 函数:

```
Server.Listener.run()
> SERVER.set(Server.this)
> connectionManager.startIdleScan()
> while (running) {           //Listener 线程的主循环
>+ getSelector().select()    //睡眠等待连接请求到来
>+ Iterator<SelectionKey> iter = getSelector().selectedKeys().iterator() //请求可能不止一个
>+ while (iter.hasNext()) {   //逐个扫描同时到来的连接请求
>++ key = iter.next()
>++ iter.remove()
>++ if (key.isValid()) if (key.isAcceptable())
>+++ doAccept(key)           //接受连接请求
>+++> ServerSocketChannel server = (ServerSocketChannel) key.channel()
```



```

>+++> while ((channel = server.accept())!= null) { //接受连接请求后成为一个通道
>+++>+ Reader reader = getReader() //从数组 readers 中获取一个空闲的 Reader 线程
>+++>+ Connection c = connectionManager.register(channel) //在此通道上建立一个连接
>+++>+ key.attach(c); // so closeCurrentConnection can get the object
>+++>+ reader.addConnection(c) //将此连接指派给这个 Reader 线程
>+++>+ pendingConnections.put(conn) //挂入这个 Reader 线程的待处理队列
>+++>+ readSelector.wakeup() //唤醒这个线程,下面就是这个线程的事了
>+++> }
>+ } // while (iter.hasNext())
> } //end while(running)

```

Listener 线程监听着它的 Socket 通道,当有连接请求到来并认为有效和可接受时,就为其创建一个 Connection 对象,把它交给一个 Reader 处理。同时到来的请求可能不止一个,所以这里需要有个内嵌的 while 循环。

所谓交给一个 Reader 线程处理,就是从线程数组 readers 中找一个空闲的 Reader 线程,把所创建的 Connection 对象挂入其 pendingConnections 队列。下面就是 Reader 线程的事了:

```

Server.Listener.Reader.run()
> doRunLoop()
>> while (running) { //Reader 线程的主循环
>>+ size = pendingConnections.size() //检查队列中是否有连接
>>+ for (int i = size; i>0; i--) { //若有,就逐个处理
>>+ Connection conn = pendingConnections.take() //从队列中获取一个连接
>>+ conn.channel.register(readSelector, SelectionKey.OP_READ, conn)
//登记此连接为有效
>>+ }
>>+ readSelector.select() //接收属于有效连接的报文
>>+ Iterator<SelectionKey> iter = readSelector.selectedKeys().iterator()
>>+ while (iter.hasNext()) { //对属于有效连接的每个报文,即每个 RPC 请求
>>+ key = iter.next()
>>+ iter.remove()
>>+ if (key.isValid() && key.isReadable()) doRead(key)
>>+> Connection c = (Connection)key.attachment()
>>+> count = c.readAndProcess() == Connection.readAndProcess()
>>+>> while (true) {
>>+>>+ ... //读入和处理报头
>>+>>+ count = channelRead(channel, data) //然后读入本次 RPC 请求的内容
>>+>>+ processOneRpc (data.array()) //处理一次 RPC
>>+>> }
>>+> if (count < 0) closeConnection(c)

```

```
>>+ } //end while (iter.hasNext())
```

```
>> } //end while(running)
```

```
> readSelector.close()
```

Reader 线程本来就在其主循环中睡眠等待,一旦被唤醒就扫描其 pendingConnections,就其中的每个连接向所属通道登记。然后,一旦有属于这些连接的 RPC 请求到来,就通过 doRead() 逐一加以处理,具体就是通过 readAndProcess() 提供服务,那就是 processOneRpc():

```
[Server.Listener.Reader.run() > doRunLoop() > doRead() > Connection.readAndProcess()
> processOneRpc()]
```

```
Connection.processOneRpc(byte[] buf)
```

```
> DataInputStream dis = new DataInputStream(new ByteArrayInputStream(buf))
```

```
> RpcRequestHeaderProto header =
```

```
    decodeProtobufFromStream(RpcRequestHeaderProto.newBuilder(), dis)
```

```
> callId = header.getCallId()
```

```
> checkRpcHeaders(header)
```

```
> processRpcRequest(header, dis)
```

```
>> Class<? extends Writable> rpcRequestClass = getRpcRequestWrapper(header.getRpcKind())
```

```
>> rpcRequest = ReflectionUtils.newInstance(rpcRequestClass, conf)
```

```
>> rpcRequest.readFields(dis)
```

```
>> Call call = new Call(header.getCallId(), header.getRetryCount(), rpcRequest, this,
    ProtoUtil.convert(header.getRpcKind(), header.getClientId().toByteArray(), traceSpan)
```

```
>> callQueue.put(call); // queue the call; maybe blocked here, 队列已满就睡眠等待
    //将此 call 挂入 Server 的 callQueue 队列,交由 Handler 处理
```

注意这里的 call 是个 Call 类对象,代表着一个 RPC 调用请求。不要与前述的 call() 函数相混淆。

像 Listener 有个线程数组 readers 一样,Server 也有个线程数组 handlers,里面有一组 Handler 线程,这些线程都是在 Server 的初始化阶段在其 start() 函数中创建的。每个 Hanler 线程都盯着 Server 的 callQueue 队列,一被唤醒就试图从这个队列里获取一个 call 加以处理。

```
Handler.run()
```

```
> SERVER.set(Server.this)
```

```
> ByteArrayOutputStream buf = new ByteArrayOutputStream(INITIAL_RESP_BUF_SIZE)
```

```
> while (running) { //Hanler 线程的主循环
```

```
>+ Call call = callQueue.take(); // pop the queue; maybe blocked here
```

```
    //试图从队列中取一个 call 对象,没有就睡眠等待
```

```
>+ if (!call.connection.channel.isOpen()) continue
```

```
>+ CurCall.set(call) //以此 call 为当前 Call
```

```
>+ if (call.connection.user == null) { //如果所属的连接是全方位的:
```

```
>++ value = call(call.rpcKind, call.connection.protocolName,
```

```
    call.rpcRequest, call.timestamp)
```

```

        //调用 RPC.Server.call(),替对方在本地实施这个过程调用,即 RPC
>+ } else { //如果所属的连接有访问权限的管理,就以对方这用户的名义和身份
>++ value = call.connection.user.doAs(new PrivilegedExceptionAction<Writable>())
        ] run()
        > return call(call.rpcKind, call.connection.protocolName,
                                call.rpcRequest, call.timestamp)
        //调用 RPC.Server.call(),替对方在本地实施这个过程调用,即 RPC
>+ }
>+ CurCall.set(null)
>+ setupResponse(buf, call, returnStatus, detailedErr, value, errorClass, error) //设置响应报文
>+ responder.doRespond(call) //对请求方做出回应:
>+> call.connection.responseQueue.addLast(call) //挂入所属连接的 responseQueue 队列
>+> if (call.connection.responseQueue.size() == 1) { //如果队列中只有这么一个回应
>+>+ processResponse(call.connection.responseQueue, true) //那就越俎代庖了
>+> }
> } // end while

```

Handler 线程专门替对方在本地实施远程过程调用,那就是这里对 call()的调用要达到的目标。至于 call()具体是怎么实施,如前所述,Server 是个抽象类,实际上这要看具体 call 所遵循的规程即 Protocol,那要由扩充落实 Server 的实体类提供,后面我们会看到这方面的实例。

实施了 call 之后,要对请求方作出回应,包括返回结果(函数的返回值有可能是个需要加以串行化的对象),所以这里要通过 setupResponse()准备好一个响应,Call 类的对象内部有个缓冲区 rpcResponse,就是用来盛放回应信息的。Handler 线程并不直接发送回应,另外有 Responder 线程专管这个事,只要把 call 挂入其所属连接的 responseQueue 队列就行。

每个连接,即每个 Connection 类对象,都有个 responseQueue 队列。而 Server 则有个 Responder 线程,这是在 Server 的构造函数中创建的。每当创建一个连接时都要向这个 Responder 线程登记,让其照看这个连接的 responseQueue 队列。

所以,Responder 线程在其主循环中监视着所有连接的 responseQueue 队列:

```

Responder.run()
> SERVER.set(Server.this)
> doRunLoop()
>> while (running) { //Responder 线程的主循环
>>+ waitPending(); // If a channel is being registered, wait
>>+ writeSelector.select(PURGE_INTERVAL)
        //等待所有已登记连接的 responseQueue 队列中有需要回应的 call 到来
>>+ Iterator<SelectionKey> iter = writeSelector.selectedKeys().iterator()
>>+ while (iter.hasNext()) { //对于其中的每一个 call
>>++ SelectionKey key = iter.next()
>>++ iter.remove()

```

```

>>++ if (key.isValid() && key.isWritable()) doAsyncWrite(key)
>>++> Call call = (Call)key.attachment()
>>++> if (call == null) return
>>++> processResponse(call.connection.responseQueue, false) //发送响应报文
>>+ }
>>+ ...
>> }

```

有关响应报文发送即 processResponse()的具体细节,这里就不深入下去了。实际上这也可以很复杂,因为网络上发送的包是有长度限制的,一个响应报文可能要好多次才能发完。但是不了解这些底层的细节不至于严重影响我们对 RPC 机制的理解。

另外,注意我们这是摘要,实际上有许多(我认为)对当前这个主题不那么重要的代码就被省略了,想要彻底搞个明白的读者还是应该回到源代码中。

对提供服务的 Server 一方有所了解以后,我们把目光转向提出服务请求的一方。请求服务的一方称为客户方,或用户方,这一边定义了一个与 Server 相对应的 Client 类,凡是对外提出服务请求都需要由 Client 对象经手。当然,Client 对象只是在 Server 面前代表客户方,而非客户方的全部。

```

class Client {} //org.apache.hadoop.ipc.Client, a client for an IPC service.
] Hashtable<ConnectionId, Connection> connections //Connection 线程的集合
//一个 Client 可以有多个对外连接,每个连接都有个 Connection 线程照看
] class ClientExecutorServiceFactory {}
] setPingInterval()
] checkResponse(RpcResponseHeaderProto header) //Check the rpc response header
] createCall(RPC.RpcKind rpcKind, Writable rpcRequest)
> return new Call(rpcKind, rpcRequest)
] class Call {} //Class that represents an RPC call,每个 Call 对象代表着一次 RPC 调用
] getConnection(ConnectionId remoteId, Call call,
int serviceClass, AtomicBoolean fallbackToSimpleAuth) //建立连接
] class Connection extends Thread {} //在 Client 这一边,每个连接都有个 Connection 线程
]] Socket socket
]] DataInputStream in
]] DataOutputStream out
]] setupConnection()
> while (true) {
>+ try {
>++ this.socket = socketFactory.createSocket()
>++ this.socket.setTcpNoDelay(tcpNoDelay)
>++ this.socket.setKeepAlive(true)
>++ NetUtils.connect(this.socket, server, connectionTimeout)
>++ this.socket.setSoTimeout(pingInterval)

```

```

> ++ return
> + }
> }

]] sendPing()
> if( curTime - lastActivity.get() >= pingInterval) {
> + out.writeInt(pingRequest.size())
> + pingRequest.writeTo(out)
> + out.flush()
> }

]] run()
> while (waitForWork()) { //wait here for work - read or close connection
> + receiveRpcResponse()
> }

]] sendRpcRequest(Call call)
]] receiveRpcResponse()
]] call(Writable param, InetSocketAddress address)
//注意 Client.call()与 Server.call()是两码事
> call(RPC.RpcKind.RPC_BUILTIN, param, address)
>> call(rpcKind, param, address, null)
>>> ConnectionId remoteId = ConnectionId.getConnectionId(addr, null, ticket, 0, conf)
>>> call(rpcKind, param, remoteId)

]] call(RPC.RpcKind rpcKind, Writable rpcRequest, ConnectionId remoteId, int serviceClass)
> call = createCall(rpcKind, rpcRequest)
> connection = getConnection(remoteId, call, serviceClass)
> connection.sendRpcRequest(call)
> while (!call.done) call.wait()

```

Client 管理着对外(服务方)的连接。凡是已经建立并且还维持着的连接,都有一个 Connection 线程,Client 类对象内部有个集合 connections,根据 ConnectionId 可以从中找到该连接的 Connection 线程。

当客户方需要进行 RPC 调用时,就通过 Client.call()发出调用请求。注意,这个 call()与前面 Server 中的那个 call()是两码事。Client 有好几个采用不同参数序列的 call()函数,其中最简单的是只有两个参数的 call():

```

call(Writable param, InetSocketAddress address)
> call(RPC.RpcKind.RPC_BUILTIN, param, address)

```

这里的 param 就是要求进行 RPC 的报文,address 当然是对方的地址。这个 call()函数添上了另一个实参 RPC.RpcKind.RPC_BUILTIN,调用另一个 call()函数。RPC.RpcKind 意为“RPC 的方式”,是个枚举类型,其取值有三种:RPC_BUILTIN、RPC_WRITABLE 和 RPC_PROTOCOL_BUFFER,这里我们暂且不必关心。

最后解决问题、真正“干实事”的是下面这个 call():


```

Client.call(RPC.RpcKind rpcKind, Writable rpcRequest,
ConnectionId remoteId, int serviceClass, AtomicBoolean fallbackToSimpleAuth)
> Call call = createCall(rpcKind, rpcRequest)    //创建一个 Call 对象
>> return new Call(rpcKind, rpcRequest)
> Connection connection = getConnection(remoteId, call, serviceClass, fallbackToSimpleAuth)
//如果尚未建立连接就与对方建立连接:
>> do {
>>+ Connection connection = connections.get(remoteId)
>>+ if (connection == null) {
>>++ connection = new Connection(remoteId, serviceClass)
>>++ connections.put(remoteId, connection)
>>+ }
>> } while (!connection.addCall(call))
>> connection.setupIOStreams(fallbackToSimpleAuth)
>> return connection
> connection.sendRpcRequest(call) == Connection.sendRpcRequest(call) //发出 RPC 请求
> while (!call.done) call.wait() //等待对方的回应
> return call.getRpcResponse() //返回对方的回应

```

进行 RPC 调用,首先要创建一个代表着这次 RPC 的 Call 对象;然后从 connections 中找到通往同一个 Server、同一类服务的 Connection 线程,如果还没有建立就加以创建;再通过该连接的 sendRpcRequest() 发送 RPC 请求。

我们不妨先看一下 Connection 类的构造函数:

```

Connection.Connection(ConnectionId remoteId, int serviceClass)
> this.remoteId = remoteId
> this.server = remoteId.getAddress()
> this.rpcTimeout = remoteId.getRpcTimeout()
> this.maxIdleTime = remoteId.getMaxIdleTime()
> this.connectionRetryPolicy = remoteId.connectionRetryPolicy
> this.maxRetriesOnSasl = remoteId.getMaxRetriesOnSasl()
//采用 Sasl 加密时允许失败的次数
> this.maxRetriesOnSocketTimeouts = remoteId.getMaxRetriesOnSocketTimeouts()
> this.tcpNoDelay = remoteId.getTcpNoDelay()
> this.doPing = remoteId.getDoPing() //需不需要定时发送 Ping 报文
> if (doPing) { // construct a RPC header with the callId as the ping called
//如果需要就预先准备好一个用于 Ping 的报头备用
>+ pingRequest = new ByteArrayOutputStream() //分配缓冲区,以供构筑 Ping 报头
>+ RpcRequestHeaderProto pingHeader =
ProtoUtil.makeRpcRequestHeader(RpcKind.RPC_PROTOCOL_BUFFER,
OperationProto.RPC_FINAL_PACKET, PING_CALL_ID,

```

```

        RpcConstants.INVALID_RETRY_COUNT, clientId)
    >+ pingHeader.writeDelimitedTo(pingRequest)
    > }
    > this.pingInterval = remoteId.getPingInterval() //Ping 的时间间隔
    > this.serviceClass = serviceClass
    > UserGroupInformation ticket = remoteId.getTicket() //对方所要求的“门票”
        // try SASL if security is enabled or if the ugi contains tokens.
        // this causes a SIMPLE client with tokens to attempt SASL
    > boolean trySasl = UserGroupInformation.isSecurityEnabled() ||
        (ticket != null && !ticket.getTokens().isEmpty())
        //表示需不需要试试 Sasl 加密
    > this.authProtocol = trySasl ? AuthProtocol.SASL : AuthProtocol.NONE;
        //如果需要就是 AuthProtocol.SASL
    > this.setDaemon(true) //将线程设置成 Daemon,关闭标准输入输出通道

```

注意,这只是创建 Connection 对象(虽然是个线程)时的构造函数,基本上都是类似于初始化的操作,尚未真正建立与 Server 方的连接。这里所设置的那些变量的名称都比较能说明问题,就不需要过多解释了。将来,建立起与 Server 方的连接之后,可能是需要定时发送 Ping 报文的,如果需要就预先准备好一个用于 Ping 的报头备用。

所得到的 Connection,不管是 connections 中原已存在的,还是新创建的,总之都要经过 setupIOstreams()这一步。不过,若是原来已经建立的连接就马上可以返回了;若是新创建的 Connection 就得建立起与 Server 方的连接。注意,这里 Connection 指一个对象,这是一个线程,而“连接”则指与 Server 方建立的网络连接。

```
[Client.call() > Connection.getConnection() > setupIOstreams()]
```

```
Connection.setupIOstreams(AtomicBoolean fallbackToSimpleAuth)
```

```

> if (socket != null || shouldCloseConnection.get()) return //与 Server 的连接业已建立
> while (true) { //新创的 Connection 需要建立与 Server 方的连接
>+ setupConnection()
>+> while (true) {
>+>+ this.socket = socketFactory.createSocket()
>+>+ UserGroupInformation ticket = remoteId.getTicket()
>+>+ if (ticket != null && ticket.hasKerberosCredentials()) {
>+>+ ... //略
>+>+ }
>+>+ NetUtils.connect(this.socket, server, connectionTimeout) //与 Server 建立网络连接
>+>+ return
>+> } //end while (true) //如果在此过程中发生异常而失败,就循环再试
>+ InputStream inStream = NetUtils.getInputStream(socket) //建立基于 socket 的输入流
>+ OutputStream outStream = NetUtils.getOutputStream(socket) //建立基于 socket 的输出流

```

```

>+ writeConnectionHeader(outStream) //写这个输出流,就是通过 socket 发送报头
>+> DataOutputStream out = new DataOutputStream(new BufferedOutputStream(outStream))
>+> out.write(RpcConstants.HEADER.array()) //4 个字节:“hrpc”
>+> out.write(RpcConstants.CURRENT_VERSION) //1 个字节的版本号
>+> out.write(serviceClass) //1 个字节的类别
>+> out.write(authProtocol.callId) //1 个字节的身份认证规程,NONE(0)或 SASL(-33)
>+ if (authProtocol == AuthProtocol.SASL) {
>+ ... //与“主旋律”无关,有兴趣或需要的读者请自行阅读
>+ }
>+ if (doPing) inStream = new PingInputStream(inStream) //在输入流上增加对 Ping 的处理
>+ this.in = new DataInputStream(new BufferedInputStream(inStream))
//把基于 socket 的原始输入流进一步改造成带缓冲的数据输入流,隐藏 socket 的本性
>+ if (!(outStream instanceof BufferedOutputStream)) {
>+> outStream = new BufferedOutputStream(outStream) //让输出流也带上缓冲
>+ }
>+ this.out = new DataOutputStream(outStream) //使输出流也变成数据输出流
>+ writeConnectionContext(remoteId, authMethod) //
>+> IpcConnectionContextProto message = ProtoUtil.makeIpcConnectionContext(
RPC.getProtocolName(remoteId.getProtocol()), remoteId.getTicket(), authMethod)
>+> RpcRequestHeaderProto connectionContextHeader =
ProtoUtil.makeRpcRequestHeader(RpcKind.RPC_PROTOCOL_BUFFER,
OperationProto.RPC_FINAL_PACKET,
CONNECTION_CONTEXT_CALL_ID,
RpcConstants.INVALID_RETRY_COUNT, clientId)
>+> RpcRequestMessageWrapper request =
new RpcRequestMessageWrapper(connectionContextHeader, message)
//创建请求报文 request
>+> out.writeInt(request.getLength()) // Write out the packet length
>+> request.write(out)
>+ touch()
>+ start() //启动这个 Connection 线程
>+ return
> }

```

这个 `setupIOStreams()`, 其实就是专为新创 Connection 线程而设的, 因为凡是原已存在的 Connection 都已经有了 socket, 所以一进来就返回了。而新创建的 Connection 线程, 则通过 `setupConnection()` 创建 socket 并与 Server 方建立网络连接。建立起网络连接之后, 还要以 socket 为基础建立起抽象程度更高的、带缓冲的数据输入流和输出流。比方说, 有了 `DataOutputStream` 就可以直接往里面写整数, 而无须关心字节的次序怎么排、怎么打成 IP 包之类的这些问题。然后, 就通过这个输出流向 Server 方发送连接请求报文。注意, 前面通过 `connect()` 建立的只是传输层的连接, 现在要建立的则是 RPC 层即应用层的连接, 是 Client 对

象与 Server 对象的连接,这是不同层次上的连接。

所以,凡是新创建的 Connection 线程,都要先与 Server 建立 RPC 连接,具体就是向其发送连接请求。回头看看 Server 一方的代码摘要可以知道,得要对方接受了连接请求之后,双方才可以进行 RPC 请求的交互。发送了 RPC 连接请求,即 ConnectionHeader 加上 ConnectionContext 以后,程序就启动 Connection 线程的运行,使其进入 run() 函数:

```
Connection.run()
> while (waitForWork()) { //wait here for work - read or close connection
    //Connection 线程的主循环
>+ receiveRpcResponse() //接收来自对方的响应
>+ int totalLen = in.readInt()
>+ RpcResponseHeaderProto header = RpcResponseHeaderProto.parseDelimitedFrom(in)
>+ checkResponse(header)
>+ int headerLen = header.getSerializedSize()
>+ headerLen += CodedOutputStream.computeRawVarint32Size(headerLen)
>+ int callId = header.getCallId()
>+ Call call = calls.get(callId)
>+ RpcStatusProto status = header.getStatus()
>+ if (status == RpcStatusProto.SUCCESS) {
>+ + Writable value = ReflectionUtils.newInstance(valueClass, conf)
>+ + value.readFields(in); // read value
>+ + calls.remove(callId)
>+ + call.setRpcResponse(value)
>+ + if (call.getRpcResponse() instanceof ProtobufRpcEngine.RpcWrapper) {
>+ + + ProtobufRpcEngine.RpcWrapper resWrapper =
    (ProtobufRpcEngine.RpcWrapper) call.getRpcResponse()
>+ + }
>+ } else { // Rpc Request failed
>+ + ... //略
>+ }
> } //end while
> close()
```

显然,在 Hadoop 的 RPC 机制中,Connection 线程只管接收来看 Server 的响应,不管发送。发送是想要进行 RPC 调用的那个线程自己的事。作为请求 RPC 的一方,Connection 接收的只能是 Server 方的回应。注意,在通过 readFields() 从响应报文中读出时会涉及对象的去串行化。

回到前面 Client.call() 的摘要中,创建了 Connection 对象(线程)并与 Server 方建立连接之后,就通过其 sendRpcRequest() 要求发送 RPC 请求:

```
[Client.call() > Connection.sendRpcRequest()]
```

```
Connection.sendRpcRequest(Call call)
```

```
> DataOutputStream d = new DataOutputStream() //准备好一块缓冲区
```

```
> RpcRequestHeaderProto header = ProtoUtil.makeRpcRequestHeader(
    call.rpcKind, OperationProto.RPC_FINAL_PACKET, call.id, call.retry,
    clientId)
```

```
> header.writeDelimitedTo(d) //把 RPC 请求报头写入缓冲区 d
```

```
> call.rpcRequest.write(d) //把 RPC 请求的内容写入缓冲区 d
```

```
> runnable = new Runnable() //创建一个 Runnable 对象(作为线程运行)
```

```
] run() //这个 Runnable 的 run() 函数
```

```
> byte[] data = d.getData() //d 就是上面准备好的缓冲区
```

```
> int totalLength = d.getLength() //这是已经写入缓冲区的长度
```

```
> out.writeInt(totalLength); // Total Length,把这个长度从输出通道 out 写出去
```

```
> out.write(data, 0, totalLength); // RpcRequestHeader + RpcRequest
```

```
> out.flush() //发送
```

```
> senderFuture.get()
```

```
> Future<?> senderFuture = sendParamsExecutor.submit(runnable)
```

```
== ExecutorService.submit(runnable) //提交给 JVM,要求安排线程执行其 run() 函数
```

这里的 out,是 Connection 线程在前面 setupIOstreams()中创建的数据输出流,其基础是 socket 和网络连接,可以将其看成通向 Server 方的输出流,现在要通过它向对方发送 RPC 请求了。发送 RPC 请求比发送连接请求麻烦,因为这可能涉及一般对象的串行化。另一方面,发送的过程中也可能会因异常而要求重试,所以这里采用异步的方式,为通过输出流 DataOutputStream 发送的过程创建一个 Runnable,并将其提交给 Java 虚拟机,让其安排线程加以执行。至于 RPC 的执行结果,则有 Connection 线程先加处理再予上交,请求 RPC 服务的线程可以在某个点上睡眠等待结果的到来。

严格说来,Server 和 Client 二者还不能说是构成了完整的 RPC 层,实质上这只是一个 IPC(进程间通信)层,或者说只是一种原始而粗糙的 RPC 底层,可以用来达到远程过程调用的目的,但是并不方便。比方说,我们确实可以通过 Client.call()调用对方某类对象的某个方法,但是这在形式上跟在本地调用同类对象的同一方法有着很大的差异,这种差异要求我们必须了解许多底层的细节,甚至可能会影响我们的思维。另一方面,这样的程序几乎注定就是不易移植、不可重用(reuse)的。设想有一天把对方的这个某类对象移到了本地,那么程序中固定写死的那些对 Client.call()的调用就都得要改成本地的调用了。所以,我们需要的是一种形式上与本地调用没有什么不同的 RPC 机制。如果我们需要调用某个远地的某类对象的某种方法,那么本地就应该有这对象的(当然是同类的)镜像,或者说“代理(proxy)”,只要调用本地的这个 proxy 的某个过程,就可以自动转化成对远地那个对象真身的过程调用。这样,一方面写程序的人再也不必关心许多细节(其实是把有关的工作集中到了少数几个人身上),思维上也可更自然通畅,而且即使有一天把这个某类对象移到了本地,绝大部分的程序也可不必更改,只要把 proxy 去掉就行了。这样的机制,才是我们想要的 RPC 机制。

还有个问题:需要被用作 RPC 目标的类及其方法可能是五花八门的,不同的类、不同的方法(函数),乃至不同的参数类型,最后实现具体调用时的程序代码也就有所不同(否则编译就

通不过)。这也是为什么前面的 Server 只能是抽象类的原因,既然要在远地的服务端以不同的参数类型和序列对不同类的不同函数进行调用,它的 call() 函数怎么能固定得下来,怎么能有统一的实现呢? 这样看来,我们是否只好“一事一议”,针对需要进行远程调用的每一个函数都准备一个特别的 call()。但是那样就增加了大量机械而枯燥无味又容易出错的工作,最好能有个软件工具来帮助生成这样的代码。还有个只适用于解释型语言的办法,就是让解释器在运行时根据每个函数的界面定义临时决定怎么调用,但是那样做一来太烦琐,容易出错,二来对运行效率也有影响。事实上,Java 语言的 Reflection 机制,就是在一定程度上保留了解释型语言的特点,本质上是把一些编译时的决定和选择推迟到了运行时。Hadoop 既然采用 Java,这就也是个不坏的方案。当然 Reflection 机制不可滥用,否则就变成解释型语言了(Java 语言从 Java 语句到虚拟机指令这一步是编译的,从虚拟机指令到物理 CPU 指令这一步则是解释的)。所以,这二者,即软件辅助生成代码和适度使用 Reflection 机制,可以互相补充,结合使用。

Hadoop 在这方面的设计,应该就是出于这样的考虑,也是从 2.0 版开始,采用了 ProtocolBuffer 以及与之配套的方案。其实,上述 Server 和 Client 的设计就已经是与这个方案配套的了,其早期的 RPC 机制不是这样的。Hadoop 的 RPC 机制在 2.0 版前后有很大的不同,下面所讲当然都是指 2.0 版及以后的 Hadoop。

Hadoop 的代码中定义了一个 RPC 类,用来帮助应用层构筑具体的 RPC 层设施,以达到上述这两方面的目标,即通过 proxy 进行 RPC 调用,以及软件辅助生成代码与 Reflection 机制的结合。我们先看 RPC 类的定义摘要:

```
class RPC {}
] enum RpcKind {
    RPC_BUILTIN ((short) 1),           // Used for built in calls by tests
    RPC_WRITABLE ((short) 2),          // Use WritableRpcEngine
    RPC_PROTOCOL_BUFFER ((short) 3);   // Use ProtobufRpcEngine
}
] Map<Class<?>, RpcEngine> PROTOCOL_ENGINES
] interface RpcInvoker {}
]] call(Server server, String protocol, Writable rpcRequest, long receiveTime)
] setProtocolEngine(Configuration conf, Class<?> protocol, Class<?> engine)
    > conf.setClass(ENGINE_PROP + "." + protocol.getName(), engine, RpcEngine.class)
    // ENGINE_PROP 就是“rpc.engine”
] getProtocolEngine(Class<?> protocol, Configuration conf)
    // 获取或创建具体的 ProtocolEngine, 有 ProtobufRpcEngine 和 WritableRpcEngine 两种
    ----- 以下用于客户端的创建 -----
] getProxy(Class<T> protocol, long clientVersion,
    InetAddress addr, Configuration conf, SocketFactory factory)
    > return getProtocolProxy(protocol, clientVersion, addr, conf, factory).getProxy()
>> if (UserGroupInformation.isSecurityEnabled()) {
    SaslRpcServer.init(conf);
```

```

>>> }
>>> pe = getProtocolEngine(protocol, conf)
>>> pe.getProxy(...) == ProtobufRpcEngine.getProxy(Class<T> protocol, long clientVersion,
    InetAddress addr, UserGroupInformation ticket,
    Configuration conf, SocketFactory factory, int rpcTimeout,
    RetryPolicy connectionRetryPolicy,
    AtomicBoolean fallbackToSimpleAuth)
>>>> Invoker invoker = new Invoker(protocol, addr, ticket, conf, factory, rpcTimeout,
    connectionRetryPolicy, fallbackToSimpleAuth) //ProtobufRpcEngine.invoker
>>>> p = Proxy.newProxyInstance(protocol.getClassLoader(), new Class[] {protocol}, invoker)
>>>> new ProtocolProxy<T>(protocol, p, false) //创建 ProtocolProxy 对象
----- 以下用于服务端的创建 -----
] class Builder {} //Class to construct instances of RPC server with specific options.
[] setProtocol(Class<?> protocol)
[] setInstance(Object instance)
[] build()
> return getProtocolEngine(this.protocol, this.conf).getServer(
    this.protocol, this.instance, this.bindAddress, this.port,
    this.numHandlers, this.numReaders, this.queueSizePerHandler,
    this.verbose, this.conf, this.secretManager, this.portRangeConfig)
] abstract static class Server extends org.apache.hadoop.ipc.Server{}

```

先要对上述摘要做些解释和说明。

首先这里引入了 protocol 的概念,所谓 protocol,当然是 Client/Server 之间的规程和协议。但请注意,这是 RPC 层的协议,不要与传输层的协议如 UDP、TCP 之类混淆。如前所述, RPC 所针对的类和函数可谓五花八门,调用界面各不相同,可是大的分类总还是有的。例如与提交作业和分配资源有关的函数就各有不少,于是我们就可以说,提交作业有提交作业的 protocol,分配资源有分配资源的 protocol。类似地, NM 节点与 RM 节点之间有 protocol, NM 节点互相之间又是另一种 protocol。在实践中,采用不同 protocol 的软件一般是由不同的人员开发的。前面说过,最好是远地有个什么类的对象、本地就有这个什么类对象的镜像即 proxy;这样当然是好,但是也失之繁杂,因为数量巨大。折中的办法是以 protocol 为单位,让每个 protocol 都有自己的 Server 和 proxy。所以,当我们说要创建一个 Server(或 proxy)时,首先要明确这是要创建哪一种 protocol 的 Server(或 proxy),需要有对于此种 protocol 的定义和说明,这样才能生成对于其中各个函数的调用(代码),并且有什么样的 Server 就有什么样的 proxy。所以,对于 RPC 机制的实现,protocol 是要素。

然后是 RpcEngine 和 ProtocolEngine 的概念。什么是 RpcEngine 呢? RpcEngine 是 Hadoop 代码中定义的一个界面:

```

interface RpcEngine {}
[] ProtocolProxy<T> getProxy(Class<T> protocol, ...) //返回一个 proxy 对象
[] RPC.Server getServer(Class<?> protocol, ...) //返回一个 RPC.Server 对象

```

```
] getProtocolMetaInfoProxy(ConnectionId connId, Configuration conf, SocketFactory factory)
```

注意, `getProxy()` 的返回类型是 `ProtocolProxy<T>`, 即某种类型的 `ProtocolProxy`。而 `getServer()` 的返回类型则是 `RPC.Server`, 即对于 `RPC.Server` 的某种落实了它的 `call()` 函数的扩充(见前述), 总之是某种类型的 `RPC.Server` 对象。

而 `ProtocolEngine`, 则是作为工作“引擎”具体实现某种 `RpcKind` 的 RPC 机制并实现了 `RpcEngine` 界面的类。在 Hadoop 中这样的引擎有两种:

```
class ProtobufRpcEngine implements RpcEngine {}
class WritableRpcEngine implements RpcEngine {}
```

就是说, 这两种引擎都可以生成 `RPC.Server` 和 `Proxy`。那区别何在呢? 前者是基于 Protobuf 的, 后者是基于 Writable 的。前者不用解释了, 那当然就是 Google 的 Protobuf。后者所谓的 Writable, 是 Hadoop 代码中定义的一个界面:

```
interface Writable {}
[ write(DataOutput out)
[ readFields(DataInput in)
```

这个界面只提供 `write()` 和 `readFields()` 这两种操作, 实际上就是任意格式的字节串。相较于 Protobuf, 这似乎是无格式的、原始形式的数据, 但是这里面实际上是存在着某种格式的, 所以就需要有运行时的解析(Parsing)。

其实还可以有第三种, 那就是前面所述的底层 `Server` 和 `Client` (当然要补上 `Server.call()` 的代码)。

为了区分具体引擎, 从而分清具体 RPC 请求的类型 (`Server` 与 `Client` 即 `proxy` 必须配套), `RPC` 类中定义了一个枚举类型 `RpcKind`, 取值范围为 `RPC_BUILTIN`、`RPC_WRITABLE` 和 `RPC_PROTOCOL_BUFFER`。其中对 `RPC_BUILTIN` 的注释说是为测试用, 但是实际上恐怕也不仅仅是测试, 凡是底层的、不纳入具体 protocol 的 RPC 调用都得用 `RPC_BUILTIN`。

现在, Hadoop 代码中定义的这个 `RPC` 类, 其设计意图应该比较容易理解了。其中 `setProtocolEngine()` 和 `getProtocolEngine()` 的意义自明, 其余就是用来生成或获取 (如果已经生成) 针对具体 protocol 的 `Server` 和 `Proxy`。

对于某个具体的 protocol, 要为其创建 `Server` 对象时, 就直接或间接调用定义于 `RPC` 类内部的 `Builder.build()`; 要为其创建 `Proxy` 对象时就调用 `RPC` 类的 `getProxy()`。而 `Builder.build()`, 实际上就是 `getProtocolEngine().getServer()`。

用 `RPC.getProxy()` 创建 `Proxy` 的实例在 Hadoop 的代码中比比皆是。随便抓一个, 比方说 `LocalizationProtocol` 的 `Proxy`:

```
public LocalizationProtocolPBClientImpl(long clientVersion,
                                         InetAddress addr, Configuration conf) throws IOException {
    RPC.setProtocolEngine(conf, LocalizationProtocolPB.class, ProtobufRpcEngine.class);
    proxy = (LocalizationProtocolPB)RPC.getProxy(
        LocalizationProtocolPB.class, clientVersion, addr, conf);
```

```
}

```

创建 Server 的操作则要多样化一些,并且往往也不是那么直截了当的。我们以 ResouceManager 节点上的 ApplicationMasterService 为例,在其初始化阶段的 serviceStart() 中是这样创建其 Server 的:

```
ApplicationMasterService.serviceStart()
> YarnRPC rpc = YarnRPC.create(conf) // YarnRPC 是抽象类,落实为 HadoopYarnProtoRPC
> ...
> this.server = rpc.getServer(ApplicationMasterProtocol.class,
                                this, masterServiceAddress, serverConf, ...)
    //ApplicationMasterService.server 的类型就是 org.apache.hadoop.ipc.Server
    //所以一定是直接或间接扩展了这个 Server 的某类对象
    == HadoopYarnProtoRPC.getServer(ApplicationMasterProtocol.class, this, ...)
>> ...
>> return RpcFactoryProvider.getServerFactory(conf).getServer(protocol,
    instance, addr, conf, secretManager, numHandlers, portRangeConfig)
    // RpcFactoryProvider.getServerFactory(conf) 返回具体的 RpcServerFactory
>>> RpcServerFactoryPBImpl.getServer()
>>>> ...
>>>>> RpcServerFactoryPBImpl.createServer(pbProtocol, addr, conf, secretManager,
    numHandlers, (BlockingService)method.invoke(null, service), portRangeConfig)
>>>>>> RPC.setProtocolEngine(conf, pbProtocol, ProtobufRpcEngine.class)
>>>>>> RPC.Server server = new RPC.Builder(conf).setProtocol(pbProtocol)
    .setInstance(blockingService).setBindAddress(addr.getHostName())
    .setPort(addr.getPort()).setNumHandlers(numHandlers).setVerbose(false)
    .setSecretManager(secretManager).setPortRangeConfig(portRangeConfig)
    .build()
    == RPC.Builder.build() //创建 RPC.Server 对象

```

顺着程序中的逐层调用,读者自不难明白,这 ApplicationMasterService 对象的 Server,尽管多层辗转,最后还是靠 RPC.Builder.build() 创建的。这里 setInstance()、setPort() 等都是 RPC.Builder 提供的方法函数,只是上面的摘要中没有列出,读者可以直接查看源代码。

再如 HDFS 子系统中的 NameNodeRpcServer,就是直接调用 RPC.Builder.build() 创建的:

```
NameNode.createRpcServer(Configuration conf)
> return new NameNodeRpcServer(conf, this)
>> NameNodeRpcServer(conf, this)
>>> ...
>>>> this.serviceRpcServer = new RPC.Builder(conf).setProtocol(
    org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolPB.class)
    .setInstance(clientNNPbService)...build()

```

读者自然要问,这两个例子中创建的两个 Server,它们的 call()函数又是什么样的呢?前面的 RPC.Server 是抽象类,为什么这里就可以创建实体的对象呢?

我们先回到 RPC 的摘要,里面还定义了一个界面 RpcInvoker。这个界面上只定义了一个函数,那就是 call()。

对这个界面的实现是在 ProtobufRpcEngine 和 WritableRpcEngine 中。

```
class WritableRpcInvoker implements RpcInvoker {}    //在 WritableRpcEngine 中
class ProtoBufRpcInvoker implements RpcInvoker {}    //在 ProtobufRpcEngine 中
```

读者想必还记得前面讲述为什么要把 RPC.Server 定义成抽象类的原因,那里说的 rpcKind 就是一个 RpcKind 枚举值,RpcInvoker 可以是这二者之一,而 RPC.Server 的 call() 函数会落实到 WritableRpcInvoker.call()或 ProtoBufRpcInvoker.call()。

先看 WritableRpcInvoker 的 call()是怎么实现的:

```
WritableRpcEngine.WritableRpcInvoker.call(org.apache.hadoop.ipc.RPC.Server server,
    String protocolName, Writable rpcRequest, long receivedTime)
> Invocation call = (Invocation)rpcRequest
> if (call.declaringClassProtocolName.equals(VersionedProtocol.class.getName())) {
>+ VerProtocolImpl highest = server.getHighestSupportedProtocol(
    RPC.RpcKind.RPC_WRITABLE, protocolName)
>+ protocolImpl = highest.protocolTarget
> } else {
>+ ...
> }
> Method method = protocolImpl.protocolClass.getMethod(
    call.getMethodName(), call.getParameterClasses())
> method.setAccessible(true)
> Object value = method.invoke(protocolImpl.protocolImpl, call.getParameters())
> return new ObjectWritable(method.getReturnType(), value)
```

这里的关键是在作为目标的类 protocolClass 中,可以根据所调用方法的名称和参数的类型序列,通过 getMethod()找到这个方法 method,再通过 method.invoke()加以调用。这就是 Java 语言的 Reflection 机制赋予程序员的手段。当然,程序员须为每个 Protocol 提供一个 protocolImpl,实现该 Protocol 中定义的所有方法函数,RPC 调用时以这些函数为跳板,先进入这些函数(之一),再设法在这些函数中转入真正的目标函数。如前所述,如果用人工编程的话,这将是—份机械而枯燥无味的工作还容易有错。

可见,这里所利用的就是 Java 语言的 Reflection 机制。一般的编译型语言,例如 C,是没有这种机制的。给你一个对象,即数据结构,要在运行时(而不是编译时)按函数名从中找到某个函数指针加以调用,或在运行时按字段名从数据结构中找到一个字段,在 C 语言中是没有这种机制的。当然你自己也可以设法实现,但是那就相当于给 C 也添上了 Reflection 机制,这可不是易事(C 后裔之一的 Go 语言就实现了这种机制)。事实上,如果用 C 语言实现类似功能的话,多半会在 Server 一方安排一个函数跳转表,即函数指针数组,Client 一方则在编译

时将函数名转换成数组下标,即目标函数的序号。这样效率当然很高,但是灵活性会差一些。

再看 `ProtoBufRpcInvoker` 的 `call()` 是怎么实现的:

```

ProtoBufRpcEngine.ProtoBufRpcInvoker.call(RPC.Server server, String protocol,
        Writable writableRequest, long receiveTime)
> RpcRequestWrapper request = (RpcRequestWrapper) writableRequest
> RequestHeaderProto rpcRequest = request.requestHeader
> String methodName = rpcRequest.getMethodName()    //目标方法的名称
> String protoName = rpcRequest.getDeclaringClassProtocolName() //Protocol 的名称
> long clientVersion = rpcRequest.getClientProtocolVersion() //Protocol 的版本号
> ProtoClassProtoImpl protocolImpl = getProtocolImpl(server, protoName, clientVersion)
        //包含着对具体 Protocol 的实现,由 ProtocolBuf 提供
> BlockingService service = (BlockingService) protocolImpl.protocolImpl
        //由 ProtocolBuf 为该 Protocol 提供的一个实现了 BlockingService 界面的某类对象
> MethodDescriptor methodDescriptor =
        service.getDescriptorForType().findMethodByName(methodName)
        //根据方法名从该类对象内部找到对目标函数的描述
> Message prototype = service.getRequestPrototype(methodDescriptor)
> Message param =
        prototype.newBuilderForType().mergeFrom(request.theRequestRead).build()
        //从调用请求 request 中恢复参数序列
> result = service.callBlockingMethod(methodDescriptor, null, param) //调用这个方法的中介
> return new RpcResponseWrapper(result)

```

显然,这里也是依靠 `Reflection`。不同的是,这里还涉及由 `ProtocolBuf` 生成提供的底层模块。后面我们将看到,对于用 `ProtoBuf` 语言编写的每个 `Protocol`,编译工具 `protoc` 会自动生成相关代码,为每个 `Protocol` 提供一个实现了 `BlockingService` 界面的无名类,这个类中有 `Protocol` 中定义的所有方法函数。

现在我们可以回到前面的两个问题,即 `ApplicationMasterService.server` 和 `NameNode` 的 `NameNodeRpcServer` 提供了什么样的 `call()` 函数。

要回答这个问题,我们需要仔细看一下 `RPC.Builder.build()` 是怎样实现的:

```

RPC.Builder.build()
> RpcEngine engine = getProtocolEngine(this.protocol, this.conf)
>> RpcEngine engine = PROTOCOL_ENGINES.get(protocol) //也许已经在这个集合中
>> if (engine == null) { //若没有,需要创建
>>+ Class<?> impl = conf.getClass(ENGINE_PROP + "." + protocol.getName(),
        WritableRpcEngine.class)
        //这就是通过 setProtocolEngine()设置的 RpcEngine: ProtoBufRpcEngine
>>+ engine = (RpcEngine)ReflectionUtils.newInstance(impl, conf)
>>+ PROTOCOL_ENGINES.put(protocol, engine) //放入 PROTOCOL_ENGINES

```

```

>>> }
>>> return engine
>>> return engine.getServer(
    this.protocol, this.instance, this.bindAddress, this.port,
    this.numHandlers, this.numReaders, this.queueSizePerHandler,
    this.verbose, this.conf, this.secretManager, this.portRangeConfig)
    == ProtobufRpcEngine.getServer(this.protocol, this.instance, this.bindAddress, ...)
>>> return new Server(protocol, protocolImpl, conf, bindAddress, port,
    numHandlers, numReaders, queueSizePerHandler, verbose, secretManager,
    portRangeConfig)//这是 ProtobufRpcEngine.Server,它扩充了 RPC.Server

```

RPC 对象内部维持着一个已知 Protocol, 例如 ApplicationMasterProtocol 那样的集合 PROTOCOL_ENGINES, 根据具体 Protocol 的名称就可以从中找到它用的 RpcEngine。如果这个集合中还没有, 那就到配置块 conf 中查找, setProtocolEngine() 就是用来设置这个配置项的。我们在前面看到, 对这几个 Protocol 都设置了 ProtobufRpcEngine, 所以这里所调用的就是 ProtobufRpcEngine.getServer(), 而 ProtobufRpcEngine.getServer() 所做的事就是创建一个 Server 对象。既然这是在 ProtobufRpcEngine 对象内部, 这里所说的 Server 当然是 ProtobufRpcEngine.Server, 此类对象的构造函数是这样的:

```

class ProtobufRpcEngine.Server extends RPC.Server {}
] Server(Class<?> protocolClass, Object protocolImpl,
    Configuration conf, String bindAddress, int port, int numHandlers,
    int numReaders, int queueSizePerHandler, boolean verbose,
    SecretManager<?extends TokenIdentifier> secretManager,
    String portRangeConfig)
> super(bindAddress, port, null, numHandlers, numReaders, queueSizePerHandler, conf,
    classNameBase(protocolImpl.getClass().getName()),
    secretManager, portRangeConfig)
> registerProtocolAndImpl(RPC.RpcKind.RPC_PROTOCOL_BUFFER,
    protocolClass, protocolImpl) //向 RPC 登记

```

这里要执行 super(), 就是其父类的构造函数。ProtobufRpcEngine.Server 是对 RPC.Server 的扩充, 所以这里执行的是 RPC.Server.Server()。我们在前面看到, RPC.Server 是对最原始的那个 Server 的扩充, 它实际上已经补上了缺失的 call() 函数, 而且整个类中再没有什么抽象成分, 然而却定义为 abstract。这就使编译器把住了关, 不让你用 new 操作创建 RPC.Server, 但在此直接调用其构造函数却是可以的。而且, 既然 ProtobufRpcEngine.Server 扩充了 RPC.Server, 又没有定义自己的 call(), 那自然就继承了 RPC.Server.call()。所以 ProtobufRpcEngine.Server.call() 就是 RPC.Server.call()。我们再回顾一下它的摘要:

```

RPC.Server.call(RPC.RpcKind rpcKind, String protocol, Writable rpcRequest, long receiveTime)
> invoker = getRpcInvoker(rpcKind) //对于 ProtobufRpcEngine, 这是 ProtoBufRpcInvoker
>>> RpcKindMapValue val = rpcKindMap.get(rpcKind)

```

```
> return invoker.call(this, protocol, rpcRequest, receiveTime)
    == ProtobufRpcEngine.ProtoBufRpcInvoker.call(this, protocol, rpcRequest, receiveTime)
```

这里形参 `rpcKind` 所对应的实参是 `RPC_PROTOCOL_BUFFER`, 所以 `invoker` 就是 `ProtoBufRpcInvoker`。而 `ProtoBufRpcInvoker.call()`, 我们在上面已经看到, 它依赖于 `ProtoBuf` 模块为具体 `Protocol` 提供的一个实现了 `BlockingService` 界面的对象。所以, 再往下就是 `ProtoBuf` 怎样提供这个对象的问题了。

再次强调, 这里创建的是 `ProtobufRpcEngine.Server`, 而不是 `RPC.Server`, 前者是对后者的扩充。

4.2 ProtoBuf

在像 Hadoop 这样大量使用 RPC 的系统中, `Protocol` 的数量应该不少, 具体目标函数的数量就更多了。对于每个具体的 `Protocol`, 有了前述的 `RPC` 类和可以用来生成 `Server` 和 `Proxy` 的 `ProtocolEngine` 后, `Client` 端的编程工作量已可大大降低, 但是 `Server` 端虽然也降低了却总少不了要有一层远程调用的入口函数, 这些入口函数的位置上本来应该就是目标函数本身, 但一般在进出目标函数的前后总还是少不了有些哪怕是很简单的处理, 所以还是需要有一层类似于“跳板”那样的中转。然而这些中转函数的编程却是份机械而枯燥的工作, 还容易出错, 维护也麻烦。

不仅如此, `ProtocolEngine` 中还应包括具体 `protocol` 所需的报文 (message) 生成和解析, 在采用面向对象语言的系统中这又包括对象的串行化 (serialize, 也称序列化) 和去串行化, 因为通过网络传输的信息必须按位串行。对象的串行化和去串行化原理上倒不复杂, 但却是相当麻烦的操作。在 2.0 以前的版本中, Hadoop 自己实现了各种 `protocol` 的入口函数和对各种 `protocol` 对象的串行化和去串行化, 整个引擎相当于现在的 `WritableRpcEngine`。但是, 从 2.0 版开始, Hadoop 改用了 Google 的 `ProtoBuf` 软件和工具来做这些事, 由此而形成的 `ProtocolEngine` 就是 `ProtobufRpcEngine`。

`ProtoBuf` 是 `Protocol Buffer` 的缩写, 原是 Google 为内部使用而开发的一个软件项目, 后来成了开源项目。这个项目旨在为 `RPC` 开发一个灵活方便的中间层, 其底层是 `IPC`, 就像由前述 `Server` 和 `Client` 构成的通信机制, 其上层则是具体的应用。这个中间层的作用, 就是在机器节点之间建立支持具体 `protocol` 的 `RPC` 机制, 在本地提供一个跟远地相同的 `API`, 并提供实现了这个 `API` 的服务器 `server` 和代理 `proxy`, 使得客户端应用层对定义于这个 `API` 的函数 (方法) 调用转化成相应的请求报文, 交由下面的 `IPC` 层发送到远地的服务端, 在远地又转化成对于那里相同 `API` 上对应函数 (方法) 的调用, 再把调用结果返回到客户端, 作为对应用层的返回结果。

为此, `ProtoBuf` 项目创造了一种 `proto` 语言并提供相应的编译器 `protoc`, 让开发者可以方便地用这种语言描述所要实现的 `protocol`, 然后通过编译自动生成相应的代码。以 `YARN` 用户与 `RM` 节点交互所用的 `ApplicationClientProtocol` 为例, 我们看一下 Hadoop 代码中的一个 `.proto` 文件 `applicationclient_protocol.proto`, 这就是为此 `protocol` 编写的描述文件:

```
option java_package = "org.apache.hadoop.yarn.proto";
option java_outer_classname = "ApplicationClientProtocol";
```

```

...
package hadoop.yarn

import "Security.proto";
import "yarn_service_protos.proto";

service ApplicationClientProtocolService {
    rpc getNewApplication (GetNewApplicationRequestProto)
        returns (GetNewApplicationResponseProto);
    rpc getApplicationReport (GetApplicationReportRequestProto)
        returns (GetApplicationReportResponseProto);
    rpc submitApplication (SubmitApplicationRequestProto)
        returns (SubmitApplicationResponseProto);
    ... //下面还有很多
}

```

这个文件定义和描述了一种称为 ApplicationClientProtocolService 的 service, 这种服务具体支持包括 getNewApplication、getApplicationReport、submitApplication 在内的多种 RPC 调用。以其中的 submitApplication 为例, 客户端在发起 RPC 调用、请求远方调用其 submitApplication() 方法时, 应向远方发送一个 SubmitApplicationRequestProto 报文(message); 对方在完成操作之后则返回一个 SubmitApplicationResponseProto 报文。这里并没有涉及这两种报文的格式, 那是在另一个文件 yarn_service_protos.proto 中定义的, 所以前面要 import "yarn_service_protos.proto"。另外, 每个报文中其实都有与信息安全有关的字段, 所以也要 import "Security.proto"。

还要解释一下这些报文的命名。以 SubmitApplicationRequestProto 为例, 这里的关键字部分是 SubmitApplicationRequest, 后缀 Proto 并不表示 SubmitApplicationRequest 就是一个 Protocol, 而只是说这是一个“Protocol 数据单元”。事实上这只是 ApplicationClientProtocol 所定义的诸多报文格式之一。

这个文件还通过“java_outter_classname=”这个语句指示编译器: 经过编译之后, 为这个 Service 生成的代码应该都放在一个名为 ApplicationClientProtocol 的外层 class 之内。另一方面, 由于这个文件只定义了一个 service, 所以在 ApplicationClientProtocol 类内部的第一层中将只直接定义一个类, 那就是“class ApplicationClientProtocolService”。在 ApplicationClientProtocolService 内部, 则将提供客户端和服务端两边的种种操作方法。

不过, 具体与报文格式有关的代码不在 ApplicationClientProtocol 类的内部, 因为如上所述那是在另一个 .proto 文件中定义的。

我们也粗略看一下 yarn_service_protos.proto:

```

option java_package = "org.apache.hadoop.yarn.proto";
option java_outter_classname = "YarnServiceProtos";
...
package hadoop.yarn;

```

```

import "Security.proto";
import "yarn_protos.proto";

...

message SubmitApplicationRequestProto {
    optional ApplicationSubmissionContextProto application_submission_context = 1;
}

...

```

这个文件中定义了许多属于 YARN 子系统的报文格式,这里只摘列了其中的一种,就是用来请求提交作业的 SubmitApplicationRequestProto。这种报文中只有一个成分,那是一个 ApplicationSubmissionContextProto。因为一共只有这么一个成分,其序号当然就是 1。经 protoc 编译之后,这个文件所定义的内容都在一个名为 YarnServiceProtos 的 class 中。

而 ApplicationSubmissionContextProto,则又是在另一个.proto 文件 yarn_protos.proto (见前面的 import 语句)中定义的:

```

option java_package = "org.apache.hadoop.yarn.proto";
option java_outer_classname = "YarnProtos";

...

package hadoop.yarn;

import "Security.proto";

...

message ApplicationSubmissionContextProto {
    optional ApplicationIdProto application_id = 1;
    optional string application_name = 2 [default = "N/A"];
    optional string queue = 3 [default = "default"];
    optional PriorityProto priority = 4;
    optional ContainerLaunchContextProto am_container_spec = 5;
    optional bool cancel_tokens_when_complete = 6 [default = true];
    optional bool unmanaged_am = 7 [default = false];
    optional int32 maxAppAttempts = 8 [default = 0];
    optional ResourceProto resource = 9;
    optional string applicationType = 10 [default = "YARN"];
    optional bool keep_containers_across_application_attempts = 11 [default = false];
    repeated string applicationTags = 12;
    optional int64 attempt_failures_validity_interval = 13 [default = -1];
    optional LogAggregationContextProto log_aggregation_context = 14;
    optional ReservationIdProto reservation_id = 15;
}

```



```
optional string node_label_expression = 16;
optional ResourceRequestProto am_container_resource_request = 17;
}
...
```

除 Security.proto 以外这个文件没有再导入别的.proto 文件,这意味着各种字段的类型除有关安全的之外均可在本文件中找到定义,直到全都被解析成如 string、bool、int32 那样的基本类型为止,实施串行化/去串行化的代码最终就是针对基本类型的。

例如这里的字段 application_id,其类型 ApplicationIdProto 就定义在同一文件中:

```
message ApplicationIdProto {
    optional int32 id = 1;           // 一个 32 位整数
    optional int64 cluster_timestamp = 2; // 一位 64 位整数
}
```

这个文件中定义的内容经编译以后全都在一个名为 YarnProtos 的 class 中。

由此可见,service 语句相当于 Java 的 interface 定义,message 语句则相当于纯数据结构的 class 定义。而编译器 protoc,则根据这些语句生成必要的代码,将 service 语句转变成实现了相应 interface 的 class 定义;将原本只相当于数据结构定义的 message 语句转变成包括操作方法在内的 class 定义。至于 Protobuf 所提供的串行化/去串行化功能,以及作为中间层的承上启下,则全都体现在所生成的这些代码中。编译器 protoc 可以生成 Java、C++、Python、Ruby 等多种语言的代码,我们在这里只关心 Java。

这样,经过 protoc 的编译,上述的三个.proto 文件就被编译成三个 Java 文件,产生了三个 class 的代码,即 ApplicationClientProtocol、YarnServiceProtos 和 YarnProtos。其中 YarnServiceProtos 和 YarnProtos 是公共的,覆盖了 YARN 这一层所定义的所有报文格式;而 ApplicationClientProtocol 则只是 YARN 的诸多 RPC 服务所用 protocol 之一。由于 message 的种类很多,YarnServiceProtos.java 和 YarnProtos.java 这两个源文件竟各有 4 万多行!显然,当报文的种类较多时,用来实现串行化/去串行化和合成/解析的代码量是相当大的,若要人工编写确实是个负担,ProtoBuf 的作用由此可见。不过我们在这里所关心的并非如何实现串行化/去串行化和合成/解析的细节,而是 RPC 机制的构建和流程。我们关心的是:服务端和客户端两边的 RPC“协议栈(Protocol Stack)”是怎样搭建起来的;进一步,以作业提交即 SubmitApplication()为例,一次完整的 RPC 操作是怎样完成的,要走过怎样一个流程。从这个角度看,ApplicationClientProtocol 这个 class 对我们远远更为重要。当然,YARN 子系统中定义的 RPC 服务很多,更不说还有 HDFS 子系统中定义的 RPC 服务,这里只是用 ApplicationClientProtocol 作为一个实例来说明问题。

我们先分层看一下 ApplicationClientProtocol 这个 class 的结构:

```
class ApplicationClientProtocol {
    ] abstract class ApplicationClientProtocolService implements com.google.protobuf.Service{}
    ] static com.google.protobuf.Descriptors.FileDescriptor descriptor;
```

ApplicationClientProtocol 是由 protoc 根据文件 ApplicationClientProtocol.proto 编译生成的外层 class。这个.proto 文件中只有一个 service 语句,所以这个 class 的内部就只有一个

成分,就是 `ApplicationClientProtocolService`,这是个抽象类。之所以是抽象类,是因为其内部对应于 `Service` 语句中各 `RPC` 子句的操作方法都还有待落实。这个抽象类必须实现 `ProtoBuf` 中定义的 `Service` 界面,这个界面上的操作方法主要是 `callMethod()`,后面我们将会看到其实现和作用。除此之外,每个这样的外层 `class` 内部都有个 `FileDescriptor`,即文件描述块;同时还有个用来读取这个描述块的方法 `getDescriptor()`,不过这里并未列出,我们也不关心。

所以 `ApplicationClientProtocolService` 是 `ApplicationClientProtocol` 的主体,实质性的内容都在这里,下面是这个抽象类的摘要:

```
abstract class ApplicationClientProtocol.ApplicationClientProtocolService
                                implements com.google.protobuf.Service{}

- - - - - 第一部分:对于两个界面的定义: - - - - -

] interface Interface {           //异步操作界面
    //定义于这个 interface 中的 22 个抽象方法同样来自 rpc 子句,都是三个参数:
]] abstract submitApplication(RpcController controller,
    GetApplicationReportRequestProto request, protobuf.RpcCallback<> done)
]] ...
] interface BlockingInterface { //同步操作界面
    //与上面这个 Interface 相对应,同样也是 22 个抽象方法,但都是两个参数
]] abstract submitApplication(RpcController controller,
    GetApplicationReportRequestProto request)
]] ...

- - - - - 第二部分:对应于 service 语句中各 rpc 子句的抽象方法 - - - - -
    //共有 22 个,这里只列出其一
    //22 个抽象方法都有同样的参数表,都是三个参数,都是面向异步操作
] abstract submitApplication(RpcController controller,
    GetApplicationReportRequestProto request, protobuf.RpcCallback<...> done)
]...

- - - - - 第三部分:对 com.google.protobuf.Service 界面上三个方法的实现 - - - - -
] callMethod(MethodDescriptor method, RpcController controller,
    Message request, RpcCallback done) //四个参数,用于异步操作
    > switch(method.getIndex()) {
    > case 2:
    >+ return this.submitApplication(controller, request, specializeCallback(done))
    > }
] getRequestPrototype(MethodDescriptor method)
    > switch(method.getIndex()) {
    > case 2:
    >+ return SubmitApplicationRequestProto.getDefaultInstance()
    > }
] getResponsePrototype(MethodDescriptor method)
```

```

> switch(method.getIndex()) {
> case 2:
>+ return SubmitApplicationResponseProto.getDefaultInstance()
> }

```

----- 第四部分:服务端 Server 的实现 -----

```

] newReflectiveService (Interface impl) //用来创建 ApplicationClientProtocolService 对象
    //注意,参数 impl 应该是个实现了上述 Interface 界面的某类对象
> return new ApplicationClientProtocolService() //在此动态落实 22 个抽象方法的定义
] submitApplication(controller, request, com.google.protobuf.RpcCallback<...> done)
    > impl.submitApplication(controller, request, done)
        //调用参数 impl,是对上述异步 Interface 的某种实现
] ... //其他抽象方法的动态定义从略
] newReflectiveBlockingService (BlockingInterface impl)
    //用来创建实现同步操作界面 protobuf.BlockingService 的对象
    //注意,参数 impl 应该是个实现了上述 BlockingInterface 界面的某类对象
> return new com.google.protobuf.BlockingService() {}
    // BlockingService 是个 interface,需要提供所定义函数的实现才能创建对象
    //所定义函数之一是个统一的 callBlockingMethod(),用作 protocol 的总入口
] callBlockingMethod(MethodDescriptor method, RpcController controller,
    Message request) //与前面 callMethod()相似,但只有 3 个参数
    > switch(method.getIndex()) {
    > case 2:
    >+ return impl.submitApplication(controller, request) //这里只有两个参数了
    > }
] getRequestPrototype(MethodDescriptor method)
] getResponsePrototype(MethodDescriptor method)

```

----- 第五部分:客户端 Stub 即 Proxy 的创建 -----

```

] newStub(protobuf.RpcChannel channel) //创建异步的 Stub 对象
    > return new Stub(channel)
] class Stub extends ApplicationClientProtocolService implements Interface {}
    //注意,Stub 所实现的正是上述的(异步)Interface 界面
]] Stub(com.google.protobuf.RpcChannel channel) //Stub 对象的构造函数
]] submitApplication()
> channel.callMethod(getDescriptor().getMethods().get(2), controller, request, ...)
]] ... //前面 Interface 中定义了更多抽象函数,分别对应于 proto 文件中的 rpc 语句
] newBlockingStub(protobuf.BlockingRpcChannel channel) //创建同步的 BlockingStub 对象
    > return new BlockingStub(channel)
] class BlockingStub implements BlockingInterface {}
    //注意 BlockingStub 所实现的正是上述的(同步)BlockingInterface 界面
]] BlockingStub(com.google.protobuf.BlockingRpcChannel channel) //构造函数

```

```

]] submitApplication()
    > return channel.callBlockingMethod(getDescriptor().getMethods().get(2), ...)
]] ...

```

这个抽象类有五个方面的内容。

第一部分是两个界面(interface)的定义,即 Interface 和 BlockingInterface,其中 Interface 是异步操作界面,BlockingInterface 是同步操作界面。这两个界面各自定义了包括 submitApplication()在内的 22 个操作方法,但是调用参数有所不同。异步界面 Interface 所定义的都是异步操作,一调用就立即返回,实际的工作交给一个独立的线程去完成,所以是不阻塞(Non-Blocking)的,但是调用者必须提供一个供回调的操作方法 done()(实质上是个函数指针),让那个线程在完成操作之后就调用这个方法,把结果返回给调用者。所以,异步界面上的那些操作方法都有三个参数。而同步界面 BlockingInterface 所定义的则都是同步操作,其所在线程在调用的过程中可能因某些操作被“阻塞”而进入睡眠等待,一直到操作完成时才被唤醒,所以是 Blocking。同步界面上的这些操作方法都只有两个参数,因为无须提供回调方法。

这两个界面是为客户端定义的,客户端的 RPC 操作都要跨节点进行,整个过程可能需要延续较长的时间,并非立即可以完成,所以才有阻塞/不阻塞,即同步/异步的问题。事实上,后面第五部分的 Stub 和 BlockingStub 这两个类分别实现了 Interface 和 BlockingInterface 这两个界面。

第二部分是包括 submitApplication()在内的 22 个抽象操作方法,分别对应着 ApplicationClientProtocol.proto 中 service 语句里面的 22 个 RPC 子句,那就是服务端,即 RM 节点上为客户端提供的与 Application 直接相关的 22 种 RPC,这是尚未落实的。

第三部分是对于 com.google.protobuf.Service 这个界面所定义的三个方法的实现,包括 callMethod()、getRequestPrototype()、getResponsePrototype()。其中 callMethod()相当于一个统一的门户,将一个方法描述块 MethodDescriptor 作为(第一个)参数交给 callMethod(),它就根据描述块中的序号和该界面上所定义方法的列表调用相应的方法,其余的参数则被用作调用相应方法时的参数。由于调用相应方法的过程同样可能很长,所以也有同步/异步的问题;而 com.google.protobuf.Service 是一个异步界面,因而也要提供回调函数。至于 getRequestPrototype()和 getResponsePrototype()就比较简单了,那只是根据具体的方法,即具体的 RPC 操作获取用作请求报文和响应报文的样板,这样的操作不需要长时间等待,是立即可以完成的,所以即使属于异步界面也无须提供回调函数。其实 ProtoBuf 还提供一个与此相应的同步界面 protobuf.BlockingService,下面我们会看到。

第四部分是对服务端 Server 即 RPC 提供者的实现。这里提供了创建两种 Server 的方法,一种是异步的,另一种是同步的。通过 newReflectiveService()可以在服务端创建一个异步的 Server,通过 newReflectiveBlockingService()则可以创建一个同步的 Server。两种手段都提供,任由开发者选用。

前面讲到客户端即 RPC 调用发起端的同步和异步。异步 RPC 调用意味着调用者线程一经调用不等结束就可返回而无须睡眠等待,此时可以安排这个线程先干点别的,RPC 结束返回时自会调用其回调函数,让调用者线程在回调函数中恢复同步。

那么服务端的同步和异步又是什么意思呢?我们在前面 IPC 层的 Server 中看到,服务端

有专门的 Call 接收线程和 Call 处理线程,其中接收比较简单,因为当一个请求到达服务端所在的节点时其操作系统底层会唤醒这个接收线程,接收线程把 Call 请求读进来就把它挂在一个队列 callQueue 中,而处理线程 Handler 则从 callQueue 摘下 Call 请求并加以处理,包括调用 RPC 目标函数,处理之后还要发回响应报文,返回 RPC 结果。然而对于 RPC 目标函数的执行可能会比较复杂,时间上也可能会比较长,那么是让这个 Handler 线程直接去调用相关的过程,在其自身的上下文中完成整个操作,还是让它交给别的线程去调用相关的过程?这就是区别,前者是同步的过程,后者是异步的过程。如果采用同步调用,那么这个 Handler 线程要等目标函数返回时才能去 callQueue 中摘取下一个 Call 请求,这样在负载比较重的时候就可能对“并发连接数”产生一些不利影响。

再看这两个方法函数的实现方式。

其中 newReflectiveService() 所创建的是一个 ApplicationClientProtocolService 类对象,但是 ApplicationClientProtocolService 是个抽象类,它的包括 submitApplication() 在内的 22 个操作方法都是抽象方法,所以这里就动态地加以定义落实。上面的摘要中只是列出了其中之一,即 submitApplication(),实际上这 22 个操作方法的落实就好像是由同一个模板印出来似的,都是转手调用 impl 这个对象中的同名操作方法。那么这个 impl 是什么呢?这是调用 newReflectiveService() 创建对象时的参数,应该是个实现了上述(第一部分中所定义)Interface 界面的某类对象,可以是通往 RPC 目标函数的跳板,也可以直接就是 RPC 目标函数,这要看具体的实现,这个类的代码当然不是 ProtoBuf 所能生成的。

调用 newReflectiveBlockingService() 时的参数 impl 则是个实现了上述 BlockingInterface 界面的某类对象。后面我们将会看到,在 Hadoop 的代码中,这实际上是一个 ApplicationClientProtocolPBServiceImpl 类的对象,不过这个类所直接实现的是 ApplicationClientProtocolPB 界面,那是对于上述 BlockingInterface 的扩充。注意,调用 newReflectiveBlockingService() 时所创建的是一个实现了 protobuf.BlockingService 界面的对象,它采用的是统一的入口函数 callBlockingMethod(),但是在这个方法内部的 switch 语句中会按具体方法在该界面上的序号调用 impl 内部的相关操作方法。以 submitApplication() 为例,在 ApplicationClientProtocolPB 即 BlockingInterface 界面上这是 2 号操作方法(从 0 开始编号),所以在 callBlockingMethod() 内的 case 语句中会转而调用 impl.submitApplication(),即 ApplicationClientProtocolPBServiceImpl.submitApplication()。

我们可以看到,newReflectiveService() 和 newReflectiveBlockingService() 这两个函数名中都有 Reflective 这个词。之所以如此,应该是如我们在前面所见,对目标函数的调用终究是利用 Java 的 Reflection 机制实现的。

最后,第五部分是为客户端准备、用来为客户端创建 stub 的。所谓 stub,词典上是“树桩”、“蒂头”的意思。其实这个词常被水管工用来表示这样的意思:一个水管上加了个接头,准备接上另一根水管通向某处,但是那根水管暂时还没有准备好,就先来个塞子把这接口塞住。这个意思被引申到软件实践上,则有(用来模拟实际情况或者留下外接手段的)“转接点”、“端接点”的意思。ApplicationClientProtocol 这个类的内部定义了两个类,即 Stub 和 BlockingStub,可以通过 newStub() 和 newBlockingStub() 分别创建异步和同步两种 stub。注意 Stub 和 BlockingStub 所实现的界面就是前面第一部分定义的那两个界面。

创建时的参数,也就是 stub 的下一层,分别是一个实现了 protobuf.RpcChannel 或

protobuf.BlockingRpcChannel 界面的某类对象。这两个界面都只定义了一个方法,就是 callMethod()或 callBlockingMethod()。

Protobuf 所说的同步和异步是仅就 ProtoBuf 这个局部而言的,所以 Protobuf 所说的异步确实就是异步,而 Protobuf 所说的同步却仍可以在总体上是异步的。我们在前面看到服务端有三个线程,这在总体上已经是异步的了。

至于 ProtoBuf 下面的 IPC 层,本意是需要由实际应用的开发者自己提供的,前面看到的 Server 和 Client 就是 IPC 层的一种实现。但是后来 Google 又搞了一个底层的项目叫 gRPC,用来与 ProtoBuf 配套,说是更能配合得天衣无缝。不过 IPC 这一层本来就不太复杂,所以 Hadoop 至少直到 2.7.1 版仍在使用自己的 IPC 层实现,那就是前面的 Server 和 Client。

其实 Hadoop 并没有照单全收由 ProtoBuf 提供的种种选择,而是从中选取它认为合适的素材,结合 Java 语言 Reflection 机制所提供的 Proxy 技术,开发出了自己的 RPC 机制。总的来说,Hadoop 采用了 ProtoBuf 所提供的报文生成/解析和串行化/去串行化功能,服务端大体上采用了 ProtoBuf 所提供的 BlockingService 界面,但是客户端采用的是 Reflection 机制所提供的 Proxy 技术。

4.3 Java 的 Reflection 机制

Reflection 这个词,一般都直接翻译成“反射”。但是实际上这个词还有反躬自省、自我认识的意思,在 Java 一类的编程语言中表示具体对象(object)在运行中认识甚至操纵其自身结构的能力。在传统的 C、C++ 这样的编译型语言中,假定一个数据结构中有五个函数指针,或者说有五个操作方法,那么编译器在编译的时候当然有关于这个数据结构的全面的信息,这些信息来自源代码中的相关声明和定义。但是一旦完成编译之后,这些信息实际上就转化成下标、地址、位移等方式隐藏和固化在可执行代码中,以后就不再有关于这个数据结构自身的描述,这些显式的信息在很大程度上就被丢弃了。这样,比方说,在程序执行的过程中(运行时),当你通过一个函数跳转表中的某个指针调用某个函数的时候,就无法知道通过这个跳转表还可以调用些什么别的函数,也不知道这个函数叫什么名称。另外,调用一个函数,或者访问一个字段,只能是根据预先(编译时)确定的地址或地址加位移的方式进行,而不能临时按函数名调用或按字段名访问。之所以如此,就是因为在编译以后程序内部缺少了认识和操纵其自身的手段和能力。但是在 Java 以及一些解释执行的语言中,这个问题就好解决了,因为解释器(虚拟机)本来就需要保留此类结构信息,这是在执行过程中要用的。实际上,在 Java 语言中,任何一种类型,无论其为 class、interface、enum、String,都有个 Class 对象,这就是对于这个类的结构描述。这样,比方说我们在运行时需要知道某个对象是否提供一个名叫“DoSomething”的方法,就可以让解释器查一下这个对象所属类的 Class 对象,看看里面有没有提供这个方法。这样的机制,就称为 Reflection 机制,因为它的核心和基础就是对于自身的认识。说“反射”也没有错,因为你通过镜子的反射可以看到你自己的面容。在 Java 语言中,JDK 的 java.lang.reflect 这个 package 就是为用户提供 Reflection 机制,把原本由解释器自用的功能开放给用户,而 Proxy 就是利用了这种机制的一项重要功能,Java 自己的 RMI 机制的实现就利用了此项功能。

Proxy,即“代理”,是 JDK 提供的一个类,可以说是专为 RMI 定制的。其机理是这样:先

为 Server 的服务定义一个界面,比方说 `ApplicationClientProtocolPB`,如果 Client 与 Server 是在同一 JVM 上,中间也没有什么操作需要插进去,那就可以直接调用 Server 在此界面上提供的函数。但是,如果 Client 与 Server 不在同一 JVM 上,或者我们需要在 Client 与 Server 之间插入一些什么操作,例如增加一些 Log 以利调试,那么我们就可以在 Client 所在的那个 JVM 上为 Server 创建一个“代理”,即 Proxy 对象,通常就称为 proxy,以 proxy 为中介就可以调用 server 上的服务。

Proxy 对象的创建一定是与 `InvocationHandler` 联系在一起的。`InvocationHandler` 是 JDK 中定义的一个界面,这个界面上只有一个操作方法,就是 `invoke()`。创建 Proxy 对象之前一定要先定义一个实现了 `InvocationHandler` 界面的类,例如 `Invoker`,并创建一个该类对象,例如 `invoker`,然后就可以创建 Proxy 对象了。创建时的参数包括其所代理的一个或几个界面,以及实现了 `InvocationHandler` 界面的对象 `invoker`。JDK 在创建 Proxy 对象时会动态(运行时)生成一个隐形的对象,这个对象(类)实现了给定界面上所定义的所有方法,但是每个方法函数的代码都是相同的,那就是把描述其自身的 `Method` 对象作为参数,连同上面传下来的调用参数一起,调用 `Invoker.invoke()`。这样,从 Client 看来跟以前的直接调用并无不同,因为 Proxy 提供了定义于界面上的所有方法,但是九九归一都跑到了 `Invoker.invoke()`,于是我们就可以在这个方法中实现把有关本次调用的信息编码发送到 Server 所在的 JVM,再由那里的相关程序加以解码,转化成对于那里的服务调用。或者,即使 Client 与 Server 是在同一 JVM 上,这也给了我们拦截和插入中间处理提供了手段。

事实上,Hadoop 的 RPC 机制在 Client 一侧就是通过 Proxy 实现的。

下面我们就来看,在前述的 Server 和 Client 以及 RPC 这个类的基础上,Hadoop 是如何利用 `ProtoBuf` 和 Proxy 搭建起自己的 RPC 机制,并以 `submitApplication()` 为例说明其内部的程序流程。

4.4 RM 节点上的 RPC 服务

我们先看具体的 RPC 服务端是怎么搭建起来的。

我们从 YARN 的主节点 RM 即 `ResourceManager` 开始,这是个带 `main()` 函数的类,所以会作为一个独立的 JVM 进程(或者说在一台独立的 Java 虚拟机上)运行。下面是 `ResourceManager` 类的摘要,我们就从其 `main()` 函数开始:

```
class ResourceManager extends CompositeService {}
] main(String argv[])
    > conf = new YarnConfiguration()
    > resourceManager = new ResourceManager()
    > hook = new CompositeServiceShutdownHook(resourceManager)
    > ShutdownHookManager.get().addShutdownHook(hook, ...)
    > resourceManager.init(conf) //会调用 ResourceManager.serviceInit()
    > resourceManager.start()
] ResourceManager() //构造函数
    > super("ResourceManager") == CompositeService("ResourceManager")
```

```

>>> super(name) == AbstractService
] serviceInit(Configuration conf)
  > ... // load core-site.xml
  > ... // load yarn-site.xml
  > this.rmLoginUGI = UserGroupInformation.getCurrentUser()
  > doSecureLogin()
  > rmDispatcher = setupDispatcher()
  > adminService = createAdminService()
  > createAndInitActiveServices()
>>> activeServices = new RMActiveServices(this) //其定义见后
>>> activeServices.init(conf) //会调用下面的 RMActiveServices.serviceInit()
  > super.serviceInit(this.conf)
] class RMActiveServices extends CompositeService {}
]] serviceInit(Configuration configuration)
  > ...
  > clientRM = createClientRMService()
>>> return new ClientRMService(this.rmContext, scheduler, this.rmAppManager, ...)
                                //创建 ClientRMService,这相当于 RM 面向客户的服务窗口
  > addService(clientRM)
  > ...
]] serviceStart()

```

从其 main() 函数开始,读者应能顺着程序的流程进入 createClientRMService(),就是 ClientRMService 类对象 clientRM 的创建,然后看这个类的 serviceStart():

```

[ResourceManager.main() > serviceInit() > createAndInitActiveServices() >
RMActiveServices.serviceInit() > createClientRMService() > ClientRMService.serviceStart()]

class ClientRMService extends AbstractService implements ApplicationClientProtocol{}
] serviceStart()
  > YarnRPC rpc = YarnRPC.create(conf) //实际创建的是 HadoopYarnProtoRPC
  > this.server = rpc.getServer(ApplicationClientProtocol.class, this, clientBindAddress, ...)
    == HadoopYarnProtoRPC.getServer(ApplicationClientProtocol.class, ...)
  > this.server.start()
  > super.serviceStart()
] getNewApplication(GetNewApplicationRequest request)
] submitApplication(SubmitApplicationRequest request)
  > ...
  > user = UserGroupInformation.getCurrentUser().getShortUserName()
  > rmAppManager.submitApplication(submissionContext, System.currentTimeMillis(), user)
] ...//还有好多,此处省略

```

RM 节点上的 ClientRMService 对象是专门为 RM 的客户,主要是 App 服务的。所以这个类提供了例如 `getNewApplication()`、`submitApplication()` 等函数,就是让客户通过 RPC 来调用的。为此它需要建立自己的 RPC 服务端,所以才调用 `YarnRPC.create()`:

```
[ResourceManager.serviceInit() > createAndInitActiveServices() > createClientRMService()
> RMActiveServices.serviceInit() > createClientRMService() > ClientRMService.serviceStart()
> YarnRPC.Create()]
```

```
abstract class YarnRPC {}
```

```
] create(Configuration conf)
```

```
> clazzName = conf.get(YarnConfiguration.IPC_RPC_IMPL)
```

```
== "org.apache.hadoop.yarn.ipc.HadoopYarnProtoRPC"
```

```
// 定义于 yarn-default.xml 中的“yarn.ipc.rpc.class”属性
```

```
> if (clazzName == null) clazzName = YarnConfiguration.DEFAULT_IPC_RPC_IMPL
```

```
// 那也是“org.apache.hadoop.yarn.ipc.HadoopYarnProtoRPC”
```

```
> return (YarnRPC) Class.forName(clazzName).newInstance()
```

```
// 创建 HadoopYarnProtoRPC 对象,那是对 YarnRPC 的扩充
```

但是具体该创建什么类的对象得要用“yarn.ipc.rpc.class”为键去配置块 `conf` 中查询,返回的结果应该是 `HadoopYarnProtoRPC`,所以这里创建的就是一个 `HadoopYarnProtoRPC` 类的对象,然后就调用其 `getServer()`:

```
[ResourceManager.serviceInit() > createAndInitActiveServices() > createClientRMService()
> RMActiveServices.serviceInit() > createClientRMService() > ClientRMService.serviceStart()
> YarnRPC.Create() > HadoopYarnProtoRPC.getServer()]
```

```
class HadoopYarnProtoRPC extends YarnRPC {}
```

```
] getServer(Class protocol, Object instance, InetSocketAddress addr, Configuration conf,
```

```
SecretManager<?extends TokenIdentifier> secretManager,
```

```
int numHandlers, String portRangeConfig)
```

```
> LOG.debug("Creating a HadoopYarnProtoRpc server for protocol " + protocol
```

```
+ " with " + numHandlers + " handlers")
```

```
> serverFactory = RpcFactoryProvider.getServerFactory(conf)
```

```
>> serverFactoryClassName =
```

```
conf.get(YarnConfiguration.IPC_SERVER_FACTORY_CLASS,
```

```
YarnConfiguration.DEFAULT_IPC_SERVER_FACTORY_CLASS)
```

```
>> return (RpcServerFactory) getFactoryClassInstance(serverFactoryClassName)
```

```
// 这是 RpcServerFactoryPBImp1
```

```
> return serverFactory.getServer(protocol, instance, addr, conf,
```

```
secretManager, numHandlers, portRangeConfig)
```

```
] getProxy(Class protocol, InetSocketAddress addr, Configuration conf)
```

为了创建 RPC 服务端,先得获取用来创建这个服务端的 `serverFactory`,这又得去配置块

conf 中查询, 这次的结果是 RpcServerFactoryPBImp, 因为默认的就是这个。

```
DEFAULT_IPC_SERVER_FACTORY_CLASS =
    "org.apache.hadoop.yarn.factories.impl.pb.RpcServerFactoryPBImp"
    //默认的 serverFactory
```

结果这个 serverFactory 就是 RpcServerFactoryPBImp, 再调用其 `getServer()` 方法, 就得到了服务端针对给定 protocol 的 RPC 层 Server。注意, 在我们现在这个情景中, 这里的参数 protocol 是 `ApplicationClientProtocol.class`。

```
[ResourceManager.serviceInit() > createAndInitActiveServices() > createClientRMService()
> RMActiveServices.serviceInit() > createClientRMService() > ClientRMService.serviceStart()
> YarnRPC.Create() > HadoopYarnProtoRPC.getServer()
> RpcServerFactoryPBImp.getServer()]
```

```
class RpcServerFactoryPBImp implements RpcServerFactory {}
```

```
] getServer(Class<?> protocol, Object instance, InetAddress addr, Configuration conf,
    SecretManager<?extends TokenIdentifier> secretManager,
    int numHandlers, String portRangeConfig)
> Constructor<?> constructor = serviceCache.get(protocol) //说不定已经有缓存了
    //在我们这个情景中, 参数 protocol 是 ApplicationClientProtocol.class
> if (constructor == null) { //如果缓存中还没有
>+ clazzName = getPbServiceImplClassName(protocol)
>+> String srcPackagePart = getPackageName(clazz) //参数 clazz 就是 protocol
>+> String srcClassName = getClassName(clazz) //ApplicationClientProtocol
>+> String destPackagePart = srcPackagePart + "." + PB_IMPL_PACKAGE_SUFFIX
    // "impl.pb"
>+> String destClassPart = srcClassName + PB_IMPL_CLASS_SUFFIX // "PBServiceImpl"
>+> return destPackagePart + "." + destClassPart
>+ pbServiceImplClazz = localConf.getClassByName(clazzName)
    //这就是 "ApplicationClientProtocolPBServiceImpl"
>+ constructor = pbServiceImplClazz.getConstructor(protocol)
    // 这是 ApplicationClientProtocolPBServiceImpl 的构造函数
>+ constructor.setAccessible(true)
>+ serviceCache.putIfAbsent(protocol, constructor) //加入缓存
> }
> Object service = constructor.newInstance(instance)
    //创建 ApplicationClientProtocolPBServiceImpl 对象
> Class<?> pbProtocol = service.getClass().getInterfaces()[0] //获取其实现的第一个界面
> Method method = protoCache.get(protocol) //先在缓存中找
> if (method == null) { //缓存中还没有, 创建并放入缓存
>+ clazzName = getProtoClassName(protocol)
```



```

>+ Class<?> protoClazz = localConf.getClassByName(clazzName)
           // 这个类是 ApplicationClientProtocolService
>+ method = protoClazz.getMethod("newReflectiveBlockingService",
                                   pbProtocol.getInterfaces()[0])
>+ method.setAccessible(true)
>+ protoCache.putIfAbsent(protocol, method) //加入缓存
> }
> blkserv = (BlockingService)method.invoke(null, service)
           == newReflectiveBlockingService.invoke(null, service)
           //调用 ApplicationClientProtocolService.newReflectiveBlockingService()
           //以创建一个实现了 protobuf.BlockingService()界面的对象
           //参数 service,即 ApplicationClientProtocolPBServiceImpl,成为其内部成分 impl
> return createServer(pbProtocol, addr, conf, secretManager, numHandlers, blkserv,
                      portRangeConfig)

```

这里, `method.invoke()` 的结果是一个 `BlockingService` 对象。`BlockingService` 是在 Google Protobuf 软件包中定义的一个界面,所以这结果实际上是个实现了 `BlockingService` 界面的某类对象。这个类是在抽象类 `ApplicationClientProtocolService` 内部动态定义的:

```

class ApplicationClientProtocol { }
] abstract class ApplicationClientProtocolService { }
]] newReflectiveBlockingService(final BlockingInterface impl)
    > return new com.google.protobuf.BlockingService()
                                   //创建一个实现了 BlockingService 界面的对象
] getDescriptorForType()
] callBlockingMethod(
    com.google.protobuf.Descriptors.MethodDescriptor method,
    com.google.protobuf.RpcController controller,
    com.google.protobuf.Message request) //补上需要实现的方法函数
    > switch(method.getIndex()) {
    > case 0: return impl.getNewApplication(controller, request)
    > case 1: return impl.getApplicationReport(controller, request)
    > case 2: return impl.submitApplication(controller, request)
           == ApplicationClientProtocolPBServiceImpl.submitApplication(controller, request)
    >> response = real.submitApplication(request)
           == ClientRMService.submitApplication(SubmitApplicationRequest request)
    > ...
    > }

```

所以,上面通过 Reflection 机制调用 `newReflectiveBlockingService()` 的结果是创建了一个实现了 `BlockingService` 界面的对象,并为其动态定义了界面所要求的操作方法,包括关键的 `callBlockingMethod()`。以后当有 RPC 请求到来的时候,服务端将首先就调用这个

callBlockingMethod()函数,然后在里面分情形调用诸如 impl.submitApplication(controller, request)等函数。这里的 impl 就是前面创建的 ApplicationClientProtocolPBServiceImpl 对象,那里面的变量 real 就是上一层 getServer() 中的参数 instance,再往上,就是 ClientRMService.serviceStart()中的 this,那就是 ClientRMService 对象本身。所以,这里对 impl.submitApplication()的调用最后就转化成对 ClientRMService.submitApplication()的调用,这是本次 RPC 调用的真正目标。

定义于 protobuf 软件包中的这个界面之所以称为 BlockingService,是因为这个界面上的函数调用将是同步的 Blocking 调用,而非异步的 NonBlocking 调用。

有了这个以后,就可以在此基础上通过 createServer()创建 RPC 服务端了:

```
[ResourceManager.serviceInit() > createAndInitActiveServices() > createClientRMService()
> RMActiveServices.serviceInit() > createClientRMService() > ClientRMService.serviceStart()
> YarnRPC.Create() > HadoopYarnProtoRPC.getServer()
> RpcServerFactoryPBImpl.getServer() > createServer()]

createServer(Class<?> pbProtocol, InetAddress addr, Configuration conf,
              SecretManager<?extends TokenIdentifier> secretManager, int numHandlers,
              BlockingService blockingService, String portRangeConfig)
> RPC.setProtocolEngine(conf, pbProtocol, ProtobufRpcEngine.class)
> builder = new RPC.Builder(conf)
> builder.setProtocol(pbProtocol) //以参数 pbProtocol 为这个 builder 所针对的 protocol
> builder.setInstance(blockingService) // instance == protobuf.BlockingService()
> builder.setBindAddress(addr.getHostName()).setPort(addr.getPort()).
    setNumHandlers(numHandlers).setVerbose(false).
    setSecretManager(secretManager).setPortRangeConfig(portRangeConfig)
> RPC.Server server = builder.build()
    //创建一个 RPC.Server,实际上是扩充了 RPC.Server 的某类对象
> LOG.info("Adding protocol " + pbProtocol.getCanonicalName() + " to the server")
> server.addProtocol(RPC.RpcKind.RPC_PROTOCOL_BUFFER, pbProtocol, blockingService)
> return server
    //这是一个扩充落实了 RPC.Server 的某类对象,同时也是一个 RPC.Server 类对象
```

这样,最后由 RM 创建的是服务端的“一条龙”,即整个协议栈:其底层是 IPC 层的 Server,它的上面是 RPC.Server;再上面是一个 RPC 层上实现了 com.google.protobuf.BlockingService 界面、由 ApplicationClientProtocolService 中的 newReflectiveBlockingService()方法加以动态定义和创建的无名类对象;再上面就是应用层的 ApplicationClientProtocolPBServiceImpl。

这样,当有提交作业的 RPC 请求到来时,在 RM 节点上的调用路径将是这样:

```
[RPC.Server > BlockingService.callBlockingMethod()
> ApplicationClientProtocolPBServiceImpl.submitApplication()
> ClientRMService.submitApplication()]
```

即 IPC 层将请求提交给 RPC 层的 RPC.Server,后者从接收到的 RPC 请求报文中恢复出有关参数,就调用 protobuf 层的 BlockingService.callBlockingMethod(),那里会调用作为跳板和中介的例如 ApplicationClientProtocolPBServiceImpl.submitApplication(),再由它调用本次 RPC 的目标函数,例如 ClientRMService.submitApplication()。

4.5 RPC 客户端的创建

再看客户端这一边,我们从应用层的 Client 这个类开始。

我们知道,在 Hadoop 的 IPC 层上有个名为 Client 的类,那是在 org.apache.hadoop.ipc 这个 package 中。注意在应用层也有个叫 Client 的类,这个类起着相当于 Shell 的作用,不过这是在 org.apache.hadoop.yarn.applications.distributedshell 这个 package 中。两个 Client 虽然类名相同,但是它们在命名空间的路径不同,在两个不同的 package 中。我们只要注意一下源文件前面的 package 语句就可分辨。正如这个 Client 在命名空间的路径所示,这是一个 distributedshell,即分布式的 Shell。

应用层的 Client 是个独立的应用,这个类有 main()函数,需要通过命令行启动,运行在一个独立的 Java 虚拟机上。在下面的叙述中,只要未作特别说明,凡是说到 Client 的都是指这个属于应用层的 Client。

Client 类的主 main()函数通过 new 创建 Client 对象,那就要执行其构造函数,这就开始了创建客户端 RPC 协议栈的历程。

Client.Client()

```
> YarnClient yarnClient = YarnClient.createYarnClient()
>> new YarnClientImpl()
>>> YarnClientImpl.serviceStart()
>>>> rmClient = ClientRMProxy.createRMProxy(getConfig(), ApplicationClientProtocol.class)
```

作为分布式 Shell 的 Client 是整个系统的 Client,但是同一个系统中可以同时有很多个 Client。我们现在用作例子的是作业提交,即 submitApplication(),那必须与 YARN 子系统,实际上是 RM 打交道。所以 Hadoop 代码中另有一个抽象类叫 YarnClient,意为专门针对 YARN 的 Client,这里通过其 createYarnClient()加以创建。当然,真正创建的是扩充了这个抽象类的实体类,即 YarnClientImpl。要创建 YarnClient 的不仅仅是作为分布式 Shell 的 Client,其他相对独立的模块要跟 RM 打交道时也得创建自己的 YarnClient。

RPC 层 Client 的问题,核心在于 Proxy,因为发送给服务端的 RPC 请求要靠 Proxy 传递。Proxy 就像联络员,像服务端派驻在客户端的代理人,所以这里要创建 Proxy。

注意,这里调用的 ClientRMProxy.createRMProxy()只有两个参数。我们看一下这个函数,以及 ClientRMProxy 这个类:

```
class ClientRMProxy<T> extends RMProxy<T> {}
] static ClientRMProxy INSTANCE = new ClientRMProxy()
] createRMProxy(Configuration configuration, Class<T> protocol)
> return createRMProxy(configuration, protocol, INSTANCE) //添加了参数 INSTANCE
```

] ...

ClientRMProxy 这个类有个静态成分 INSTANCE,这是个预先创建好的 ClientRMProxy 对象。而这个带两个参数的 createRMProxy()则在后面添上 INSTANCE 为第三个参数,调用带三个参数的 createRMProxy()。然而,我们在 ClientRMProxy 这个类的代码中找不到带这么三个参数的 createRMProxy(),那就去它的父类 RMProxy 中寻找,里面确实有这么一个 createRMProxy():

```
[Client.Client() > YarnClient.createYarnClient() > YarnClientImpl.serviceStart()
> ClientRMProxy.createRMProxy() > RMProxy.createRMProxy()]

RMProxy.createRMProxy(Configuration configuration, Class<T> protocol, RMProxy instance)
    //参数 protocol 实际上是界面 hadoop.yarn.api.ApplicationClientProtocol
> RetryPolicy retryPolicy = createRetryPolicy(conf)
> if (HAUtil.isHAEnabled(conf)) { //如果开通 HA,就要考虑发送失败时的对策
>+ provider = instance.createRMFailoverProxyProvider(conf, protocol)
    == RMProxy.createRMFailoverProxyProvider(Configuration conf, Class<T> protocol)
>+> defaultProviderClass = Class.forName(
    YarnConfiguration.DEFAULT_CLIENT_FAILOVER_PROXY_PROVIDER)
    //默认为 ConfiguredRMFailoverProxyProvider
>+> clazz = conf.getClass(YarnConfiguration.CLIENT_FAILOVER_PROXY_PROVIDER,
    defaultProviderClass, RMFailoverProxyProvider.class)
    //如果 conf 中没有关于“yarn.client.failover-proxy-provider”的设置,就采用默认
>+> RMFailoverProxyProvider<T> provider = ReflectionUtils.newInstance(clazz, conf)
    //创建一个 ConfiguredRMFailoverProxyProvider 对象
>+> provider.init(conf, (RMProxy<T>) this, protocol)
    //这个 this 就是前面的 ClientRMProxy.INSTANCE,是个 ClientRMProxy
>+> return provider
>+ return (T) RetryProxy.create(protocol, provider, retryPolicy)
    //以 RMFailoverProxyProvider 为基础创建高层的 RetryProxy
    == RetryProxy.create(Class<T> iface, FailoverProxyProvider<T> proxyProvider,
    RetryPolicy retryPolicy)
>+> loader = proxyProvider.getInterface().getClassLoader()
>+> invoker = new RetryInvocationHandler<T>(proxyProvider, retryPolicy)
>+> return Proxy.newProxyInstance(loader, new Class<?>[] { iface }, invoker)
> } else { //不开通 HA,就相对简单一些
>+ InetAddress rmAddress = instance.getRMAddress(conf, protocol)
>+ LOG.info("Connecting to ResourceManager at " + rmAddress)
>+ T proxy = RMProxy.<T>getProxy(conf, protocol, rmAddress)
    //先创建一个普通的 proxy,详见后
>+ return (T) RetryProxy.create(protocol, proxy, retryPolicy) //将此 proxy 包装成 RetryProxy
```

```

== RetryProxy.create(Class<T> iface, T implementation, RetryPolicy retryPolicy)
>+> failover = new DefaultFailoverProxyProvider<T>(iface, implementation)
//implementation == proxy, 为其创建一个 DefaultFailoverProxyProvider
>+> RetryProxy.create(iface, failover, retryPolicy) //跟上面有 HA 时的调用方法相同
//以 DefaultFailoverProxyProvider 为基础创建高层的 RetryProxy
== RetryProxy.create(Class<T> iface, FailoverProxyProvider<T> proxyProvider,
RetryPolicy retryPolicy)
>+>> loader = proxyProvider.getInterface().getClassLoader()
>+>> invoker = new RetryInvocationHandler<T>(proxyProvider, retryPolicy)
>+>> return Proxy.newProxyInstance(loader, new Class<?>[] { iface }, invoker)
//创建一个面向 ApplicationClientProtocol 界面的 java.lang.reflect.Proxy
> }

```

这里分两种情况：一种是开通了 HA，即高可用性（High Availability）的容错 RPC 机制；另一种是常规的 RPC 机制。不管开不开通 HA 机制，由 `RMProxy.createRMProxy()` 创建的都是一个 `RetryProxy` 对象，这个 `RetryProxy` 实质上就是一个 `Proxy`，只不过是以一个 `FailoverProxyProvider`（更确切地说是实现了此种界面的对象）为基础的 `Proxy`，所以通信失败后可以“Retry”，即重试。

在 Java 语言的 Reflection 机制中，`Proxy` 是可以嵌套的。比方说，我们可以为界面 `Interface` 创建一个 `proxy1`，其 `InvocationHandler` 是 `handler1`；然后以 `proxy1` 为基础，再为同一个界面 `Interface` 创建一个 `proxy2`，其 `InvocationHandler` 是 `handler2`。这相当于在 `proxy1` 的外面包上了一层 `proxy2`。这样，运行时首先调用 `proxy2` 所提供的定义于界面 `Interface` 的某个函数，这就到了 `handler2.invoke()`；然后在 `handler2.invoke()` 中进行了某些处理之后就调用 `proxy1` 中的同名函数，于是就又到了 `handler1.invoke()`。这样，依此类推，就可以为这个界面 `Interface` 建立起一个“协议栈”，其中处于外层的 `Proxy` 就相当于协议栈中的高层。

而 `RetryProxy`，就相当于一个两层协议栈中的高层，其底层 `Proxy` 就是与前面 `Server` 端底层相对应的客户端底层。所以，在创建高层的 `RetryProxy` 之前，先要创建底层的常规 `Proxy`。之所以需要有这么一个两层的协议栈，是为了在底层 `Proxy` 通信失败时可以由高层的 `RetryProxy` 处理重试。不过，在创建 `RetryProxy` 时不是直接以底层 `Proxy` 为基础，而是以一个 `FailoverProxyProvider`（更确切地说是一个实现了 `FailoverProxyProvider` 界面的某类对象，例如 `DefaultFailoverProxyProvider`）为基础，这个对象中包含了底层 `Proxy`，或者换言之也可以说这是把底层的 `Proxy` 包装成了 `RetryProxy`。

在开通或不开通 HA 机制的协议栈中，底层 `Proxy` 基本上是一样的，但是处理失败重发的机制却不一样。开通 HA 机制时的 `FailoverProxyProvider` 可以通过配置文件设定，如果未作设定就默认为 `ConfiguredRMFailoverProxyProvider`，而在不开通 HA 时，则采用 `DefaultFailoverProxyProvider`，这个 `FailoverProxyProvider` 处理出错重发但不支持 `Proxy` 倒换。

在不开通 HA 的情况下，用作底层 `Proxy` 的并不是前面的 `ClientRMProxy.INSTANCE`，即 `ClientRMProxy`，而是要通过 `RMProxy.getProxy()` 另外创建一个 `Proxy`，那就是一个 `ApplicationClientProtocolPBClientImpl` 类对象：


```
[Client.Client() > YarnClient.createYarnClient() > YarnClientImpl.serviceStart()
> ClientRMProxy.createRMProxy() > RMProxy.createRMProxy() > RMProxy.getProxy()]

RMProxy.getProxy(Configuration conf, Class<T> protocol, InetSocketAddress rmAddress)
> return UserGroupInformation.getCurrentUser().doAs(new PrivilegedAction<T>())
    ] run()
    > yarnRPC = YarnRPC.create(conf)
    >> clazzName = conf.get(YarnConfiguration.IPC_RPC_IMPL) //“yarn.ipc.rpc.class”
    >> if (clazzName == null) clazzName = YarnConfiguration.DEFAULT_IPC_RPC_IMPL
        //默认为“org.apache.hadoop.yarn.ipc.HadoopYarnProtoRPC”
    >> return (YarnRPC) Class.forName(clazzName).newInstance()
    > return (T) yarnRPC.getProxy(protocol, rmAddress, conf)
        == HadoopYarnProtoRPC.getProxy(protocol, rmAddress, conf)
```

创建 Proxy,以及创建了 Proxy 以后通过其进行的 RPC 操作,涉及用户的身份和权限,所以创建 Proxy 的操作是放在 doAs()方法中的一种授权行为,需要为此另建一个线程,在这个线程的 run()函数中进行。本书后面将专门详述包括 doAs()在内的安全机制。

与创建服务端的协议栈一样,这里也要先创建一个 YarnRPC 对象。不过 YarnRPC 是个抽象类,所以实际创建的对象是对 YarnRPC 的某种扩充。至于具体属于什么类则可以在配置块 conf 中加以设置,所以这里首先寻找对于“yarn.ipc.rpc.class”的设置,如果没有设置就采用默认的 HadoopYarnProtoRPC。事实上,Hadoop 的代码中扩充了 YarnRPC 的类也只有 HadoopYarnProtoRPC 这么一种。创建了 HadoopYarnProtoRPC 对象之后,就调用其 getProxy()方法获取或创建具体的 Proxy。

```
[Client.Client() > YarnClient.createYarnClient() > YarnClientImpl.serviceStart()
> ClientRMProxy.createRMProxy() > RMProxy.createRMProxy() > RMProxy.getProxy()
> HadoopYarnProtoRPC.getProxy()]

HadoopYarnProtoRPC.getProxy(protocol, rmAddress, conf)
> LOG.debug("Creating a HadoopYarnProtoRpc proxy for protocol " + protocol)
> factory = RpcFactoryProvider.getClientFactory(conf)
>> clientFactoryClassName = conf.get(YarnConfiguration.IPC_CLIENT_FACTORY_CLASS,
    YarnConfiguration.DEFAULT_IPC_CLIENT_FACTORY_CLASS)
    //默认为 RpcClientFactoryPBImpl
> return factory.getClient(protocol, 1, addr, conf)
    == RpcClientFactoryPBImpl.getClient(protocol, 1, addr, conf)
>> constructor = cache.get(protocol) //看看 cache 中是否已经有相应的构造方法
>> if (constructor == null) { //如果还没有,就创建
>>+ clazzName = getPBImplClassName(protocol) //生成针对该 protocol 的 Client 类名
    //在我们这个情景中是“ApplicationClientProtocolPBClientImpl”
>>+ pbClazz = localConf.getClassByName(clazzName)
```

```

//通过 Reflection 机制获取该类的 Class 对象
>>>+ constructor = pbClazz.getConstructor(Long.TYPE,
        InetAddress.class, Configuration.class) //获取该类的构造方法
>>>+ cache.putIfAbsent(protocol, constructor) //将此构造方法加入对照表
>>> }
>>> retObject = constructor.newInstance(clientVersion, addr, conf)
        //构造目标对象,在我们这个情景中是 ApplicationClientProtocolPBClientImpl
>>> return retObject //返回该对象

```

粗略看一下这段摘要,就会产生一个疑问:说是 getProxy(),可是实际上要获取或创建的好像是 Client 啊?其实 Proxy 和 Client 都是相对的,这里要创建的是 RPC 客户端协议栈。在客户端的应用层即真正的 Client 面前,这个协议栈的顶层代表着远地的服务端,是服务端的代理 Proxy,这一层以下是看不见的,而在这个协议栈的底层看来,其顶层却代表着应用层,是 Client,至于真正应用层的 Client 是根本看不见的。

跟创建服务端的 RPC 层 Server 一样,这里也要先通过 RpcFactoryProvider 创建一个 RpcClientFactory,具体是什么 RpcClientFactory 也是在运行时根据配置动态确定的,如果没有加以配置则默认为 RpcClientFactoryPBImp,这显然是专为采用 ProtoBuf 而设计的。创建了 RpcClientFactoryPBImp 之后就调用其 getClient()以获取或创建 RPC 层的 Client。

RpcClientFactoryPBImp 对象内部有个作为缓存 cache 的对照表,如果已经为某个 Protocol 即 Interface 创建过 Proxy,在对照表中就可以查到其相应的构造方法,那就省事了。如果是第一次创建,则要为具体的 Protocol,在我们这个情景中是 ApplicationClientProtocol,生成相应的 Client 类名,在我们这个情景中就是 ApplicationClientProtocolPBClientImpl。

生成了目标类名之后,运用 Java 的 Reflection 机制就可以获取这个类的构造函数,这里一方面把它放进作为 cache 的便查表,一方面就用这个构造函数来创建和构造 ApplicationClientProtocolPBClientImpl 对象。下面是 ApplicationClientProtocolPBClientImpl 这个类的摘要,我们主要关心两件事。

首先,这个类的对象内部有个成分 Proxy,在 ApplicationClientProtocolPBClientImpl 这里代表着 RPC 的 Server 一方,这就是客户端 RPC 协议栈中的下一层次。这个 Proxy 应该是个实现了 ApplicationClientProtocolPB 界面的某类对象,而 ApplicationClientProtocolPB 实际上就是我们在前面看到过的 ApplicationClientProtocolService.BlockingInterface,这是在 Hadoop 的源码文件 ApplicationClientProtocolPB.java 中定义的:

```

interface ApplicationClientProtocolPB
    extends ApplicationClientProtocolService.BlockingInterface{//空白}

```

这里说 ApplicationClientProtocolPB 是个扩充,可是实际扩充的内容却是空白,所以二者其实是等价的。至于 ApplicationClientProtocolService.BlockingInterface 的定义,则是 protoc 根据 proto 文件编译生成的。

其次,这个 Proxy 是在 ApplicationClientProtocolPBClientImpl 的构造函数中创建的,我们必须顺着这个创建的过程才能知道这究竟是个什么对象。

```

[Client.Client() > YarnClient.createYarnClient() > YarnClientImpl.serviceStart()

```

```
> ClientRMProxy.createRMProxy() > RMProxy.createRMProxy() > RMProxy.getProxy()
> HadoopYarnProtoRPC.getProxy() > RpcClientFactoryPBImpl.getClient()
> ApplicationClientProtocolPBClientImpl()]
```

```
class ApplicationClientProtocolPBClientImpl implements ApplicationClientProtocol{
} ApplicationClientProtocolPB proxy
] ApplicationClientProtocolPBClientImpl(long clientVersion,
    InetAddress addr, Configuration conf)    //构造函数
> RPC.setProtocolEngine(conf, ApplicationClientProtocolPB.class, ProtobufRpcEngine.class)
    //将 ApplicationClientProtocolPB 与 ProtobufRpcEngine 绑定
> proxy = RPC.getProxy(ApplicationClientProtocolPB.class, clientVersion, addr, conf)
```

由此可见, ApplicationClientProtocolPBClientImpl 内部的 Proxy 是通过 RPC.getProxy() 创建的, 所创建的是一个支持 ApplicationClientProtocolPB 界面的 Proxy。注意这里通过 RPC.setProtocolEngine 把这个界面和 ProtobufRpcEngine 绑定在一起, 这一点下面马上就要用到:

```
[Client.Client() > YarnClient.createYarnClient() > YarnClientImpl.serviceStart()
> ClientRMProxy.createRMProxy() > RMProxy.createRMProxy() > RMProxy.getProxy()
> HadoopYarnProtoRPC.getProxy() > RpcClientFactoryPBImpl.getClient()
> ApplicationClientProtocolPBClientImpl() > RPC.getProxy()]

RPC.getProxy(Class<T> protocol, long clientVersion,
    InetAddress addr, Configuration conf)
> protoProxy = RPC.getProtocolProxy(protocol, clientVersion, addr, conf)
    //三个参数, 这里的 protocol 是 ApplicationClientProtocolPB
>> sockFactory = NetUtils.getDefaultSocketFactory(conf)
    //默认 hadoop.net.StandardSocketFactory, 用来创建使用 ip 地址的 Socket
>> getProtocolProxy(protocol, clientVersion, addr, conf, sockFactory) //增加了一个参数
>>> protoEngine = getProtocolEngine(protocol, conf)
    //已与 ApplicationClientProtocolPB 绑定的是 ProtobufRpcEngine, 见前
>>>> protoEngine.getProxy(protocol, clientVersion, ...)
    == ProtobufRpcEngine.getProxy(protocol, clientVersion, ...)
>>>>> invoker = new Invoker(protocol, addr, ticket, conf, factory, rpcTimeout,
    connectionRetryPolicy, fallbackToSimpleAuth)
    //创建一个 ProtobufRpcEngine.Invoker
>>>>> proxy = Proxy.newProxyInstance(protocol.getClassLoader(),
    new Class[] {protocol}, invoker)
    //这个 Proxy 是 java.lang.reflect.Proxy
>>>>> return new ProtocolProxy<T>(protocol, proxy, false) //以 proxy 为参数之一创建
>>>>>> this.protocol = protocol
```

```
>>>>> this.proxy = proxy
> return protoProxy.getProxy() //所返回的就是刚创建的 Proxy
>> return ProtocolProxy.proxy
```

上面这段摘要让人感觉太复杂、太弯弯绕绕了,但这都是为了保持灵活性和通用性。

由于前面绑定了 ProtobufRpcEngine,这里的 Proxy 是通过 ProtobufRpcEngine.getProxy()创建的。所创建的是一个以 ProtobufRpcEngine.Invoker 为 InvocationHandler、支持 ApplicationClientProtocolPB 界面即 ApplicationClientProtocolService.BlockingInterface 界面的 ProtocolProxy 对象,其核心是一个 java.lang.reflect.Proxy 对象。这样,凡是对 Proxy 所支持界面的函数调用都会被引导到 ProtobufRpcEngine 的 Invoker.invoke()。

可是,在我们至此所见到的代码中,这个 RPC 协议栈仍未“落地”,还没有与 IPC 层挂上。这个问题是在 ProtobufRpcEngine.Invoker 的构造函数中解决的:

```
class ProtobufRpcEngine implements RpcEngine {}
[] ClientCache CLIENTS = new ClientCache()
[] class Invoker implements RpcInvocationHandler {}
[] Client client //注意,这是 IPC 层的 Client 类
[] Invoker(Class<?> protocol, InetAddress addr,
           UserGroupInformation ticket, Configuration conf, ...)
           //构造方法,8 个参数
> connId = Client.ConnectionId.getConnectionId(addr, protocol, ticket,
           rpcTimeout, connectionRetryPolicy, conf)
> this(protocol, connId, conf, factory) //转而调用下面这 4 个参数的构造方法
[] Invoker(Class<?> protocol, Client.ConnectionId connId,
           Configuration conf, SocketFactory factory)
> this.remoteId = connId
> this.client = CLIENTS.getClient(conf, factory, RpcResponseWrapper.class)
  == ClientCache.getClient()
>> Client client = clients.get(factory)
>> if (client == null) {
>>+ client = new Client(valueClass, conf, factory) //创建 IPC 层的 Client 对象
>>+ clients.put(factory, client)
>> }
>> return client
> this.protocolName = RPC.getProtocolName(protocol)
> this.clientProtocolVersion = RPC.getProtocolVersion(protocol)
```

可见,上面在创建 Invoker 对象的时候在其构造函数中创建了 Client 对象,这就是 IPC 层的 Client。我们可以看一下 ProtobufRpcEngine.java 这个文件前面的 package 语句,就可知道这是在 org.apache.hadoop.ipc 这个 package 中。这样,这个客户端的 RPC 协议栈就“落地”了,与 IPC 层挂上了钩。

回到前面 RMProxy.createRMProxy()中的 else 部分,即未开通 HA 的那个分支,我们已

经创建了那里的 Proxy, 下面就是在所创建的 Proxy 外面包上一层对于失败重发的处理, 使二者合在一起成为一个 RetryProxy, 而这个 RetryProxy 就成为 RPC 协议栈的上层 ApplicationClientProtocolPBClientImpl 所看到的 Proxy。我们在前面看到, RetryProxy 的 InvocationHandler 是 RetryInvocationHandler, 所支持的界面则是 ApplicationClientProtocol, 所以凡是对此界面上的函数调用, 例如 submitApplication() 等等, 都会被引导到这个 RetryInvocationHandler 的 invoke():

```
class RetryInvocationHandler<T> implements RpcInvocationHandler {
    ] RetryInvocationHandler(FailoverProxyProvider<T> proxyProvider, RetryPolicy retryPolicy)
                                                                    //构造方法
    > this(proxyProvider, retryPolicy, Collections.<String, RetryPolicy>emptyMap())
    >> this.proxyProvider = proxyProvider
    >> this.defaultPolicy = defaultPolicy
    >> this.methodNameToPolicyMap = methodNameToPolicyMap
    >> this.currentProxy = proxyProvider.getProxy()
    ] invoke(Object proxy, Method method, Object[] args) //发起 RPC 调用
    > RetryPolicy policy = methodNameToPolicyMap.get(method.getName())
    > if (policy == null) policy = defaultPolicy
    > boolean isRpc = isRpcInvocation(currentProxy.proxy)
    > while (true) {
    >+ if (isRpc) Client.setCallIdAndRetryCount(callId, retries) //设置是否应该重试的判定标准
    >+ try {
    >++ Object ret = invokeMethod(method, args)
    >++ return ret //如果成功就返回, 发生异常才有循环
    >+ } catch (Exception e) {
    >++ RetryAction action = policy.shouldRetry(e, retries++,
        invocationFailoverCount, isIdempotentOrAtMostOnce) //判断是否应该重试
    >++ if (action.action == RetryAction.RetryDecision.FAIL) {
    >+++ throw e //已经无可挽救, 放弃
    >+++ } else { // retry or failover
    >++++ if (action.delayMillis > 0) Thread.sleep(action.delayMillis) //等待一会儿
    >++++ if (action.action == RetryAction.RetryDecision.FAILOVER_AND_RETRY) {
    >+++++ if (invocationAttemptFailoverCount == proxyProviderFailoverCount) {
    >++++++ proxyProvider.performFailover(currentProxy.proxy) //需要倒换就倒换
    //在 HA 未开通时这是 DefaultFailoverProxyProvider.performFailover(), 是个空函数
    >++++++ proxyProviderFailoverCount++
    >+++++ } else {
    >++++++ LOG.warn("A failover has occurred since the start of this method invocation attempt."
    >+++++ )
    >+++++ currentProxy = proxyProvider.getProxy() //换一个 Proxy 试试
    >+++++ invocationFailoverCount++
```

```

>+++ } //if (action.action == RetryAction.RetryDecision.FAILOVER_AND_RETRY)
>+ } //end if - else 中的重试
>+ } //end catch
> } //end while

```

在 `invoke()` 函数中,真正把 RPC 请求发送出去要靠 `invokeMethod()`,这里把它放在一个 `while` 循环中。如果发送成功就返回;但是倘若在发送过程中因失败而发生异常,则因为这个 `while` 循环而可以反复进行努力。是否继续努力的决定则取决于 `RetryPolicies`。如果决定继续努力,则有两条路可走:一条是重试,即 `retry`;另一条是 `Failover`,即倒换,详见本书后面对于容错的介绍。

未开通 HA 时的 `proxyProvider` 是 `DefaultFailoverProxyProvider`,它的 `performFailover()` 是个空函数,所以只有在开通了 HA 的情况下这个函数才有意义。

至于 `RMPProxy.createRMPProxy()` 中开通了 HA 的那个分支,则限于篇幅不再深入下去,留给读者自己进一步阅读分析了。

至此,服务和客户两边的 RPC 协议栈都搭建好了,下面我们就以作业提交操作为例看一下具体 RPC 的流程。当用户在 Hadoop 上启动执行一个 App 作业的时候,需要把作业提交给 RM 节点。用户的 App 可以在集群中的任何一个节点上,是另起一个 Java 虚拟机、作为一个独立的 JVM 进程运行的。另外,和我们一般所理解的“运行”不同,这个进程所做的其实只是把作业提交给 RM,然后过一会儿就询问一下作业的进展,有消息就在屏幕上显示,直到作业完成。真正的计算并非在本地进行,而是分布在集群中的许多节点上。读者在下一章中将看到,App 通过一个 `YARNRunner` 对象的 `submitJob()` 方法提交作业:

```
[JobSubmitter.submitJobInternal() > YARNRunner.submitJob()]
```

```

class YARNRunner implements ClientProtocol {
] ResourceMgrDelegate resMgrDelegate
] submitJob(JobID jobId, String jobSubmitDir, Credentials ts)
    > appContext = createApplicationSubmissionContext(conf, jobSubmitDir, ts)
    > applicationId = resMgrDelegate.submitApplication(appContext)
    == ResourceMgrDelegate.submitApplication(appContext)
    >> client.submitApplication(appContext) //client 由 YarnClient.createYarnClient() 创建
    == YarnClientImpl.submitApplication(appContext)

```

读者在下一章中将看到,`YARNRunner.submitJob()` 可以说是流程中的一个关键节点,实际上可以看作是作业提交流程的起点。从这个节点开始,流程的第一站是对于 `ResourceMgrDelegate.submitApplication()` 的调用,`YARNRunner` 中的 `resMgrDelegate` 是个 `ResourceMgrDelegate` 对象,这是在 `YARNRunner` 的构造方法中创建的。然后,`ResourceMgrDelegate` 中 `client` 的类型则是 `YarnClient`,是通过 `YarnClient.createYarnClient()` 创建的,实际的类型是 `YarnClientImpl`。`YarnClientImpl` 是对 `YarnClient` 的扩充,对于客户端对服务端的通信起着桥头堡似的作用,YARN 子系统中所有 NM 对 RM 的请求都是经过这个类型的对象出去的。而 `YarnClientImpl.submitApplication()` 则是作业提交流程中

的又一个关键节点:

```
[YARNRunner.submitJob() > YarnClientImpl.submitApplication()]
```

```
YarnClientImpl.submitApplication(ApplicationSubmissionContext appContext)
```

```
>> applicationId = appContext.getApplicationId()
>> request = Records.newRecord(SubmitApplicationRequest.class)
>> request.setApplicationSubmissionContext(appContext)
>> rmClient.submitApplication(request)    //交给 rmClient 完成
>> while (true) {
>>+ try {
>>++ appReport = getApplicationReport(applicationId)
>>++> request = Records.newRecord(GetApplicationReportRequest.class)
>>++> request.setApplicationId(appId)
>>++> response = rmClient.getApplicationReport(request)    //交给 rmClient 完成
                        == ApplicationClientProtocolPBClientImpl.submitApplication(request)
>>++ state = appReport.getYarnApplicationState()
>>++ if (!state.equals(YarnApplicationState.NEW)
                        && !state.equals(YarnApplicationState.NEW_SAVING)) {
>>+++ LOG.info("Submitted application " + applicationId)
>>+++ break
>>+++ }
>>++ Thread.sleep(submitPollIntervalMillis)
>>+ } catch (ApplicationNotFoundException ex) {
>>++ // FailOver or RM restart happens before RMStateStore saves ApplicationState
>>++ LOG.info("Re-submit application " + applicationId + "with the " +
                        "same ApplicationSubmissionContext")
>>++ rmClient.submitApplication(request)
>>+ }
>> }
```

这里的 rmClient 实际上是个 ApplicationClientProtocolPBClientImpl, 这个类实现了 ApplicationClientProtocol 界面, 所以在 YarnClientImpl 的代码中对 rmClient 的类型说明是 ApplicationClientProtocol。这个类是由 protobuf 提供的, 其代码来自工具 protoc 对 proto 文件的编译:

```
[YARNRunner.submitJob() > YarnClientImpl.submitApplication()]
```

```
> ApplicationClientProtocolPBClientImpl.submitApplication()]
```

```
ApplicationClientProtocolPBClientImpl.submitApplication(SubmitApplicationRequest request)
```

```
> requestProto = ((SubmitApplicationRequestPBImp) request).getProto()
```

```
> proto = proxy.submitApplication(null, requestProto)
```

```
> return new SubmitApplicationResponsePBImpl(proto)
```

这里的 proxy,我们回顾一下,是前面在创建 ApplicationClientProtocolPBClientImpl 对象时在其构造函数中调用 RPC.getProxy() 创建的一个 ProtocolProxy 类对象。调用时的第一个参数是 ApplicationClientProtocolPB.class, 这是一个界面,说是扩充,实际上就等价于 ApplicationClientProtocolService.BlockingInterface。而 ProtocolProxy 类的核心则是由 Java 语言提供的 java.lang.reflect.Proxy。

所以,简而言之,这个 proxy 是一个以 ProtobufRpcEngine.Invoker 为 InvocationHandler,实现了 ApplicationClientProtocolService.BlockingInterface 界面的 java.lang.reflect.Proxy 对象。这样,凡是对定义于这个界面上的函数调用都被引导到 ProtobufRpcEngine.Invoker.invoke(),而 submitApplication() 正是这个界面上定义的函数之一。

```
[YARNRunner.submitJob() > YarnClientImpl.submitApplication()
> ApplicationClientProtocolPBClientImpl.submitApplication()
=> ProtobufRpcEngine.Invoker.invoke()]
```

```
ProtobufRpcEngine.Invoker.invoke(Object proxy, Method method, Object[] args)
> RequestHeaderProto rpcRequestHeader = constructRpcRequestHeader(method)
> theRequest = (Message) args[1]
> wrapper = new RpcRequestWrapper(rpcRequestHeader, theRequest)
> val = (RpcResponseWrapper) client.call(RPC.RpcKind.RPC_PROTOCOL_BUFFER,
                                         wrapper, remoteId, fallbackToSimpleAuth)
    == Client.call(RPC.RpcKind rpcKind, Writable rpcRequest,
                  ConnectionId remoteId, AtomicBoolean fallbackToSimpleAuth)
>>> call(rpcKind, rpcRequest, remoteId, RPC.RPC_SERVICE_CLASS_DEFAULT,
        fallbackToSimpleAuth)
>>>> Call call = createCall(rpcKind, rpcRequest)
>>>> Connection connection = getConnection(remoteId, call, serviceClass,
        fallbackToSimpleAuth)
>>>> connection.sendRpcRequest(call)
>>>> while (!call.done) call.wait()
>>>> return call.getRpcResponse()
>>>>> return rpcResponse //see setRpcResponse(Writable rpcResponse)
> prototype = getReturnProtoType(method)
> returnMessage = prototype.newBuilderForType().mergeFrom(val.theResponseRead).build()
> return returnMessage
```

到了 IPC 这一层,就无须多说了。

这个 RPC 请求转化为一个 ApplicationClientProtocol 协议的报文,被发送到 RM 节点。

客户端的 RPC 报文到达服务端后,由 Server.Handler 线程加以处理。从系统结构的角度看,Handler 线程处于接收端“协议栈(protocol stack)”的底部,是最靠近硬件设备的;但是从服务端的函数调用路径看,它却又是处于顶部,服务端因此而引起的一系列操作都是在

Handler 线程的上下文中由其直接或间接的调用而发生的。所以,我们就从 Handler 的 run() 函数看起。

```

Server.Handler.run()
> while (running) {
> + call = callQueue.take(); // pop the queue; maybe blocked here
> + value = call(call.rpcKind, call.connection.protocolName, call.rpcRequest, call.timestamp)
      == ProtobufRpcEngine.call(call.rpcKind,
                                call.connection.protocolName, call.rpcRequest, call.timestamp)
> +> RpcRequestWrapper request = (RpcRequestWrapper) writableRequest
> +> RequestHeaderProto rpcRequest = request.requestHeader
> +> String methodName = rpcRequest.getMethodName() //从报文中获取方法名称
> +> String protoName = rpcRequest.getDeclaringClassProtocolName() //获取协议名称
> +> ProtoClassProtoImpl protocolImpl = getProtocolImpl(server, protoName, clientVersion)
      //根据协议名称找到实现该协议的 Proto 对象
> +> BlockingService service = (BlockingService) protocolImpl.protocolImpl
      //这就是当初在 newReflectiveBlockingService()里创建的那个对象
      //在那里的代码中动态定义了一个实现 BlockingService 界面的类
> +> MethodDescriptor methodDescriptor =
      service.getDescriptorForType().findMethodByName(methodName)
      //根据方法名称找到这个方法的 MethodDescriptor
> +> prototype = service.getRequestPrototype(methodDescriptor)
> +> param = prototype.newBuilderForType().mergeFrom(request.theRequestRead).build()
> +> result = service.callBlockingMethod(methodDescriptor, null, param) //调用这个方法
      == ApplicationClientProtocol.BlockingService.callBlockingMethod(
                                                methodDescriptor, null, param)
> +>> switch(method.getIndex()) {
> +>> case 2:
> +>> + return impl.submitApplication(controller, (SubmitApplicationRequestProto)request)
      //impl 是调用 newReflectiveBlockingService()时的参数
      //就是 RM 节点上提供对 Client 服务的 ClientRMService 对象
> +>> +> ApplicationClientProtocolPBServiceImpl.submitApplication(RpcController arg0,
      SubmitApplicationRequestProto proto)
> +>> +>> request = new SubmitApplicationRequestPBImpl(proto)
> +>> +>> response = real.submitApplication(request) //这就是此次 RPC 真正的目标函数
      == ClientRMService.submitApplication(SubmitApplicationRequest request)
> +>> }
> + setupResponse(buf, call, returnStatus, detailedErr, value, errorClass, error)
> + responder.doRespond(call)
> }

```

结合注释和前文所述,这里已经很清楚,就不用多说了。

上面讲的是 YARN 子系统中提供对 App 即 Client 服务的 RPC 机制,其 protocol 是 ApplicationClientProtocol。但那只是众多服务协议中的一种,其他的还有 MRClientProtocol、ContainerManagementProtocol、DatanodeProtocol、InterDatanodeProtocol、NamenodeProtocol、ClientDatanodeProtocol、ClientNamenodeProtocol 等。但是,不管在应用层上是什么 protocol,它们的底层都是一样的,都是建立在 Protobuf 和 RPC 的基础上,并且都是采用 Java 的 reflection 机制实现的。

HDFS 子系统也是一样,也要在服务端和客户端建立起 RPC 通信的“协议栈”,也是那样的 RPC.Server 和 Client,只不过所用的 protocol 当然不是 ApplicationClientProtocol,而是用于 HDFS 的 protocol 了,但是底层的机制还是一样的。

以 NameNode 上的 NameNodeRpcServer 为例,那是 NameNode 在初始化的过程中创建的,这个类的数据结构部分的摘要如下:

```
class NameNodeRpcServer implements NamenodeProtocols {}
] FSNamesystem namesystem
] NameNode nn
] RPC.Server serviceRpcServer
] InetSocketAddress serviceRPCAddress
] RPC.Server clientRpcServer
] InetSocketAddress clientRpcAddress
```

NameNode 在其初始化阶段创建了 NameNodeRpcServer 对象,其构造函数的摘要为:

```
[NameNode.initialize() > createRpcServer() > NameNodeRpcServer.NameNodeRpcServer()]
```

```
NameNodeRpcServer.NameNodeRpcServer(Configuration conf, NameNode nn)
> int handlerCount = conf.getInt(DFS_NAMENODE_HANDLER_COUNT_KEY, ...)
> RPC.setProtocolEngine(conf, ClientNamenodeProtocolPB.class, ProtobufRpcEngine.class)
> clientProtocolServerTranslator =
    new ClientNamenodeProtocolServerSideTranslatorPB(this)
> clientNNPbService = ClientNamenodeProtocol.newReflectiveBlockingService(
    clientProtocolServerTranslator)
> dnProtoPbTranslator = new DatanodeProtocolServerSideTranslatorPB(this)
> dnProtoPbService = DatanodeProtocolService.newReflectiveBlockingService(
    dnProtoPbTranslator)
> namenodeProtocolXlator = new NamenodeProtocolServerSideTranslatorPB(this)
> NNPbService = NamenodeProtocolService.newReflectiveBlockingService(
    namenodeProtocolXlator)
> refreshAuthPolicyXlator =
    new RefreshAuthorizationPolicyProtocolServerSideTranslatorPB(this)
> refreshAuthService = RefreshAuthorizationPolicyProtocolService
    .newReflectiveBlockingService(refreshAuthPolicyXlator)
```

```

> refreshUserMappingXlator =
    new RefreshUserMappingsProtocolServerSideTranslatorPB(this)
> refreshUserMappingService = RefreshUserMappingsProtocolService.
    newReflectiveBlockingService(refreshUserMappingXlator)
> refreshCallQueueXlator = new RefreshCallQueueProtocolServerSideTranslatorPB(this)
> refreshCallQueueService = RefreshCallQueueProtocolService.
    newReflectiveBlockingService(refreshCallQueueXlator)
> genericRefreshXlator = new GenericRefreshProtocolServerSideTranslatorPB(this)
> genericRefreshService = GenericRefreshProtocolService.newReflectiveBlockingService(
    genericRefreshXlator)
> getUserMappingXlator = new GetUserMappingsProtocolServerSideTranslatorPB(this)
> getUserMappingService = GetUserMappingsProtocolService.newReflectiveBlockingService(
    getUserMappingXlator)
> haServiceProtocolXlator = new HAServiceProtocolServerSideTranslatorPB(this)
> haPbService = HAServiceProtocolService.newReflectiveBlockingService(
    haServiceProtocolXlator)
> traceAdminXlator = new TraceAdminProtocolServerSideTranslatorPB(this)
> traceAdminService = TraceAdminService.newReflectiveBlockingService(
    traceAdminXlator)
> WritableRpcEngine.ensureInitialized()
> InetSocketAddress serviceRpcAddr = nn.getServiceRpcServerAddress(conf)
> if (serviceRpcAddr != null) {
>+ String bindHost = nn.getServiceRpcServerBindHost(conf)
>+ if (bindHost == null) bindHost = serviceRpcAddr.getHostName()
>+ serviceHandlerCount = conf.getInt(DFS_NAMENODE_SERVICE_HANDLER_COUNT_KEY, ...)
>+ this.serviceRpcServer = new RPC.Builder(conf)
    .setProtocol(...ClientNamenodeProtocolPB.class)...build()
>+ // Add all the RPC protocols that the namenode implements
>+ DFSUtil.addPBProtocol(conf, HAServiceProtocolPB.class,
    haPbService, serviceRpcServer)
>+> RPC.setProtocolEngine(conf, protocol, ProtobufRpcEngine.class)
>+> server.addProtocol(RPC.RpcKind.RPC_PROTOCOL_BUFFER, protocol, service)
    == RPC.Server.addProtocol(RPC.RpcKind.RPC_PROTOCOL_BUFFER, protocol, service)
>+>> registerProtocolAndImpl(rpcKind, protocolClass, protocolImpl)
>+>>> String protocolName = RPC.getProtocolName(protocolClass)
>+>>> long version = RPC.getProtocolVersion(protocolClass)
>+>>> pn = new ProtoNameVer(protocolName, version)
>+>>> pi = new ProtoClassProtoImpl(protocolClass, protocolImpl)
>+>>> pmap = getProtocolImplMap(rpcKind)
>+>>> pmap.put(pn, pi)

```

```

>+ DFSUtil.addPBProtocol(conf, NamenodeProtocolPB.class,
                                NNPbService, serviceRpcServer)
>+ DFSUtil.addPBProtocol(conf, DatanodeProtocolPB.class,
                                dnProtoPbService, serviceRpcServer)
>+ ...
>+ // We support Refreshing call queue here in case the client RPC queue is full
>+ DFSUtil.addPBProtocol(conf, RefreshCallQueueProtocolPB.class,
                                refreshCallQueueService, serviceRpcServer)
>+ DFSUtil.addPBProtocol(conf, GenericRefreshProtocolPB.class,
                                genericRefreshService, serviceRpcServer)
>+ ...
>+ // Update the address with the correct port
>+ InetAddress listenAddr = serviceRpcServer.getListenerAddress()
>+ serviceRPCAddress = new InetAddress(serviceRpcAddr.getHostName(),
                                        listenAddr.getPort())
>+ nn.setRpcServiceServerAddress(conf, serviceRPCAddress)
> } else {
>+ serviceRpcServer = null
>+ serviceRPCAddress = null
> }
>
> InetAddress rpcAddr = nn.getRpcServerAddress(conf)
> String bindHost = nn.getRpcServerBindHost(conf)
> if (bindHost == null) bindHost = rpcAddr.getHostName()
> LOG.info("RPC server is binding to " + bindHost + ":" + rpcAddr.getPort())
> this.clientRpcServer = new RPC.Builder(conf)
                                .setProtocol(...ClientNamenodeProtocolPB.class)...build()
> // Add all the RPC protocols that the namenode implements
> DFSUtil.addPBProtocol(conf, HAServiceProtocolPB.class,
                                haPbService, clientRpcServer)
> DFSUtil.addPBProtocol(conf, NamenodeProtocolPB.class,
                                NNPbService, clientRpcServer)
> DFSUtil.addPBProtocol(conf, DatanodeProtocolPB.class,
                                dnProtoPbService, clientRpcServer)
> ...
> if (serviceAuthEnabled = conf.getBoolean(...HADOOP_SECURITY_AUTHORIZATION, false)){
>+ clientRpcServer.refreshServiceAcl(conf, new HDFSPolicyProvider())
>+ if (serviceRpcServer != null) {
>++ serviceRpcServer.refreshServiceAcl(conf, new HDFSPolicyProvider())
>+ }

```



```
> }  
> // The rpc - server port can be ephemeral... ensure we have the correct info  
> InetAddress listenAddr = clientRpcServer.getListenerAddress()  
> clientRpcAddress = new InetAddress(rpcAddr.getHostName(), listenAddr.getPort())  
> nn.setRpcServerAddress(conf, clientRpcAddress)  
> minimumDataNodeVersion = conf.get(  
    ...DFS_NAMENODE_MIN_SUPPORTED_DATANODE_VERSION_KEY, ...)  
> // Set terse exception whose stack trace won't be logged  
> this.clientRpcServer.addTerseExceptions(SafeModeException.class,  
    FileNotFoundException.class, ...)
```

看似很大一块,那只是 NameNodeRpcServer 需要支持的协议比较多而已。这就留给读者自己慢慢看了。

第 5 章

Hadoop 作业的提交

在计算机上启动运行一个应用,首先要把这个应用作为“作业(Job)”提交给计算机系统。一般这是通过键入一个命令行或点击某个图标而实现的,操作很简单。但是,如果我们要考察在提交作业时系统内部的流程,那就比较复杂了。学过操作系统的人对单机上的作业提交过程会有比较深入的了解,不过那不是本书所关注的问题。本书所关注的是,在通常运行于计算机集群的 Hadoop 系统上,作业是怎样提交的。

5.1 从“地方”到“中央”

如前所述,在 Hadoop 集群的 YARN 框架中,ResourceManager 所在的节点起着“中央”的作用,这个节点统管着整个集群的计算(HDFS 文件系统的运行另当别论),其余的节点则都处于相当于“地方”的地位。但是作业并非一定得在“中央”节点上才能提交,而完全可以从某个“地方”节点上提交。所以,作为典型的情景,作业提交的流程始自“地方”节点,但是会被提交给“中央”,然后由“中央”根据作业本身的要求(比方说需要有多少个 Mapper、多少个 Reducer,它们的数据从哪来,等等)和当时集群中的具体情况确定应该将此作业安排到哪些节点上去运行,并将其投运。不过,本章所关心的只是将作业提交到“中央”的过程,其余都是后面各章要关心的事。

这意味着,作业提交的过程涉及“地方”和“中央”两个节点上的程序流程。实际上这当然也涉及两个节点之间的 IPC 通信,但是我们现在可以不关心底层的细节。这样,作业提交的流程就自然分成了两段,我们不妨称之为“地方流程”和“中央流程”。而其中的第一段流程,就是在“地方”上的那一段,由于 Hadoop 提供了多种提交作业的途径,即多种具体的 API(应用程序设计界面),而又可进一步分成两个阶段。这里先做点简单的介绍。

“地方流程”的第一阶段由三个平行的分支构成,这就是在 Hadoop 上提交作业的三种方法,就好像地铁站有三个入口一样。这三种方法最后都汇聚在一起,于是就开始了“地方流程”的第二阶段。

从汇聚点开始的“地方流程”第二阶段,是作业提交流程的主体。在这一段流程中,“地方”要为具体的作业向“中央”打报告,形成一个作业请求,并把有关这个作业的相关材料随同报送“中央”。这段流程要到把所有这些材料都发送到“中央”才算完成。

然后就是“中央”的事了。因为本章讲的只是作业的提交,所以这段“中央流程”至少在概念上是比较简单的,只是为这个作业“立案”,并将其排入一个队列,这就完成了作业的提交。至于对这个作业请求的处理,那是后面调度和指派的事了。

下面我们先讲作业提交的第一段流程,即“地方流程”的第一阶段。

如上所述,Hadoop 为程序员提供了三种提交作业的方法,提供了三种这样的 API。之所以有三种不同的方法,是因为 Hadoop 在其历史上曾经提供了新、老两种 API,此外还提供了另一种变通的方法。这三种方法是:

(1) `JobClient.runJob()`:调用由 `JobClient` 类提供的方法 `runJob()`,这是所谓老 API。

(2) `Job.waitForCompletion()`:调用由 `Job` 类提供的方法 `waitForCompletion()`,属于新 API。

(3) `ToolRunner.run()`:调用由 `ToolRunner` 类所提供的方法 `run()`。这是一种变通的方法。

然而,光是知道有这么三种方法,并知道各自的界面定义(函数调用的参数表),还不足以向 Hadoop 正确提交一个作业。这是因为:所谓提交一个作业,主要就是提交一对 Mapper 和 Reducer,这有可能是由系统提供的,也可能是由用户(程序员)开发的,实际的应用中以后者为主,所以这也牵涉到如何设计和实现具体的 Mapper 和 Reducer 的问题。而且,在提交具体的作业时,必须填写一份类似于申请表那样的东西。填写过各种申请表的人都知道,那常常是很令人沮丧的事情,最好能有示例和样板。

正因为如此,Hadoop 的源码中提供了不少关于 MapReduce 编程的示例,这些示例自然也可用来作为 Hadoop 作业提交的示例。这些示例的源码基本上都在源码包内的目录 `hadoop-mapreduce-project/hadoop-mapreduce-examples` 中。这些示例都在教我们怎样进行 MapReduce 编程和作业提交。同时,对于我们来说更重要的是,可以用这些示例作为情景分析的起点,来分析一个作业在 Hadoop 平台上从提交到完成的完整过程。

其实不光是这个专门的目录下面有好多 MapReduce 编程示例,Hadoop 源码中还有许多别的测试程序,这些测试程序在一定程度上都可以用作编程示例。此外,Hadoop 源码中还有一些工具性的、中间层次的类,这些类对于用户而言是 Hadoop 的一部分,但是对于 Hadoop 的基础平台而言却又类似应用并且可以直接启动执行,这样的类也可以用作示例。

既然提交作业即应用程序(App)的方法有三种,我们就通过三个不同的示例来分析在 Hadoop 上提交具体作业的流程。

5.2 示例一:采用老 API 的 `ValueAggregatorJob`

通过 `JobClient` 类的 `runJob()` 方法提交作业,属于 Hadoop 的老 API。Tom White 为此在其 *Hadoop: The Definitive Guide* 一书的第二章中给出了一个示例——`OldMaxTemperature`。以前在老版本的 Hadoop 源码中也曾经有过好几个类似的示例,然而在新版本,即 2.0 版以后的 Hadoop 源码中已经难以找到采用这种方法的示例了。但是 `JobClient` 这个类还在,`runJob()` 这个方法还在,老的 API 还在,想要直接使用这个方法也还是可以的。事实上,新版的 Hadoop 源码中至今(至少到 2.7.1 版)仍在 `./hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop` 下面保留了 `mapreduce` 和 `mapred` 两个分支,后者就是采用老 API 的,其中有一个可以直接作为进程运行的类 `ValueAggregatorJob`,我们正好把它用作示例。其实 Hadoop 2.6.0 版的源码中在上述的 `mapreduce` 和 `mapred` 两个分支上分别有一个 `ValueAggregatorJob` 类。我们在这里采用的是 `mapred` 分支上的那一个。

Hadoop 源码的目录结构嵌套很深,通向具体源码文件的路径往往很长,有时书页上的一个整行还容不下一个完整的路径,读者还不如用类名搜索一下(例如在 Linux 上用 find 命令)反倒方便。Hadoop 源码的文件名很规则,一个类(嵌套在别的类内部的不算)就是一个文件,类名是什么,文件名就是什么。

我们知道,Aggregate 是“汇总、合计”的意思,所以这个类的功用一定与此有关。这个类的源码文件是 ValueAggregatorJob.java,在 mapred 分支上的 lib/aggregate 目录中,其摘要大致是这样的:

```
class ValueAggregatorJob {}  
] createValueAggregatorJobs()  
] createValueAggregatorJob()  
] setAggregatorDescriptors()  
] main()
```

在 Java 程序中,如果一个类提供了方法 main(),这个类就是“有源”的,可以从命令行直接启动作为一个 Java 虚拟机(JVM)进程运行,这跟 C/C++ 程序的情况是相似的。

注意,这个类中并没有提供相应的 Mapper 和 Reducer,所以一定来自外部,但是其源码一般都会在同一目录中,编译之后也会被打在同一个 Java 包(package)中。事实上,在这个源码文件的开头有这么一行:

```
package org.apache.hadoop.mapred.lib.aggregate;
```

而同一目录中还有好几个源码文件,其中就有 ValueAggregatorMapper.java 和 ValueAggregatorReducer.java,而且这些源码文件的头部也都指定属于同一个 package。

可想而知,如果这个应用是用户自行开发的,那么用户一般就得提供自己的 Mapper 和 Reducer。复杂一些也可能还需要提供 Combiner,这就要视具体情况而定了。Combiner 用于 Reducer 之前的数据整合。

凡是提供了 main()的类,Java 虚拟机在装载和初始化了这个类之后就会执行其 main()方法。所以我们就从 ValueAggregatorJob 的 main()开始。

```
[ValueAggregatorJob.main()]
```

```
public static void main(String args[]) throws IOException {  
    JobConf job = ValueAggregatorJob.createValueAggregatorJob(args);  
    JobClient.runJob(job);  
}
```

这里的方法(函数)名那一行最后有“throws IOException”,那是用于出错处理的,表示在这个函数的执行过程中可能发生类型为 IOException 的异常,但是函数内部没有加以捕捉和处理,所以这个函数的调用者应该加以捕捉和处理。IOException 也是一个类,是由 Java 语言提供的,凡要使用这个机制的源文件需要在文件头部加上“import java.io.IOException”。对于由 main()产生的异常,Java 虚拟机自会加以捕捉和处理。以后,为了集中我们的注意力,也为了减小篇幅,我们在遇到 throws 什么异常的时候也许就用省略号代替,或者连省略号也省略

不用了,因为一般而言这与主旋律没有什么关系。

这个 `main()` 方法中先通过本类内部提供的 `createValueAggregatorJob()` 方法创建一个 `JobConf` 类对象 `job`, 然后以此为参数调用 `JobClient.runJob()`。注意, 这里的 `JobClient` 只是一个类, 而不是一个具体的对象, 但是我们仍可直接调用这个类所提供的方法。这个类是在文件头部通过 `import` 语句导入的, 不过我们在这里并未创建任何具体的 `JobClient` 对象。

我们先看 `createValueAggregatorJob()` 做了些什么。

Java 是面向对象的程序设计语言, 由于其多态性, 在 `ValueAggregatorJob` 这个类中其实有好几个 `createValueAggregatorJob()`, 它们的调用参数各不相同, 编译器会根据调用参数确定实际调用的是哪一个方法。显然, 这里的调用参数就是 `args`, 即命令行参数的数组。

```
[ValueAggregatorJob.main() > createValueAggregatorJob(args)]
```

```
public static JobConf createValueAggregatorJob(String args[]) throws IOException {
    return createValueAggregatorJob(args, ValueAggregator.class);
}
```

这个方法只做了一件事, 就是调用另一个同名的方法。但是, 此时调用参数表不同了, 现在有两个参数, 在原有参数的后面添上了 `ValueAggregator.class`。在同一个 `Package` 的代码中搜索一下, 可知 `ValueAggregator` 是个 `interface`, 就是“界面”, 或者称为“接口”, 而 `ValueAggregator.class` 是由 Java 编译器生成的对于 `ValueAggregator` 这个界面或类的描述, 这种描述本身就是一个 `Class` 类的对象。

接下去看那个有两个参数的 `createValueAggregatorJob()`。为了帮助读者阅读, 我在源码中加上了一些中文注释。

```
[ValueAggregatorJob.main() > createValueAggregatorJob() > createValueAggregatorJob()]
```

```
public static JobConf createValueAggregatorJob(String args[], Class<?> caller)...
{
    Configuration conf = new Configuration();           //创建一个 Configuration 类对象 conf
    GenericOptionsParser genericParser = new GenericOptionsParser(conf, args);
                                                //用来解析命令行参数, 其中 args[0] 是应用的类名本身
    args = genericParser.getRemainingArgs(); //获取命令行中其余的参数(除 args[0] 以外)

    if (args.length < 2) { //后面至少还要有两个参数, 即输入目录和输出目录
        System.out.println("usage: inputDirs outputDir "
            + "[numOfReducer [textinputformat|seq [specfile [jobName]]]]");
        GenericOptionsParser.printGenericCommandUsage(System.out);
        System.exit(1);
    }

    String inputDir = args[0]; //输入目录, 注意, 原来的 args[1] 现在变成了 args[0]
    String outputDir = args[1]; //输出目录
    int numOfReducers = 1; //先假定只有一个 Reducer
```

```

    if (args.length > 2) {
        numOfReducers = Integer.parseInt(args[2]); //若命令行中另有规定就加以调整
    }

    Class<?extends InputFormat> theInputFormat = TextInputFormat.class;
    if (args.length > 3 && args[3].compareToIgnoreCase("textinputformat") == 0) {
        theInputFormat = TextInputFormat.class; //允许在命令行中指定的输入格式
    } else {
        theInputFormat = SequenceFileInputFormat.class; //默认的输入格式
    }

    ... //处理其余的命令行参数,对我们不太重要,故而从略

    JobConf theJob = new JobConf(conf); //将 Configuration 对象扩展成 JobConf 对象
    if (specFile != null) {
        theJob.addResource(specFile);
    }

    ... //这里会用到参数 caller,用于确定应该装载何种 Java 类
        //而且 caller 也可以是空(null),如果是空就强制为 ValueAggregatorJob
    theJob.setJobName("ValueAggregatorJob: " + jobName);

    FileInputFormat.addInputPaths(theJob, inputDir); //设置输入路径
    theJob.setInputFormat(theInputFormat); //设置输入格式

    theJob.setMapperClass(ValueAggregatorMapper.class); //设置 mapper 类
    FileOutputFormat.setOutputPath(theJob, new Path(outputDir)); //设置输出目录路径
    theJob.setOutputFormat(TextOutputFormat.class); //设置输出格式
    theJob.setMapOutputKeyClass(Text.class); //设置 mapper 输出的键类型
    theJob.setMapOutputValueClass(Text.class); //设置 mapper 输出的值类型
    theJob.setOutputKeyClass(Text.class); //设置最后(reducer)输出的键类型
    theJob.setOutputValueClass(Text.class); //设置最后(reducer)输出的值类型
    theJob.setReducerClass(ValueAggregatorReducer.class); //设置 reducer 类
    theJob.setCombinerClass(ValueAggregatorCombiner.class); //设置 combiner 类
    theJob.setNumMapTasks(1); // mapper 只需一个
    theJob.setNumReduceTasks(numOfReducers); //reducer 的个数取决于命令行参数
    return theJob;
}

```

先看调用参数。“Class<?> caller”是个模版式类型说明,表示任何类都可以,所以也称为“泛型”,只要是 Class 就行,在这里实际上是对于 ValueAggregator 界面的描述,包括其名称。

这个函数的代码,重点在于对 JobConf 类对象 theJob 的设置。JobConf 类是对 Configuration 类的扩充,其中的对象就是关于某个具体作业的参数配置块。我们可以把这个对象更多地看成数据结构,对这个数据结构的内容进行设置,就好像是在填写申请表或作业单。在 Hadoop 的术语中,作业(Job)就是对某个应用(Application)的可执行映像的一次运行。

在这个作业单中,有些信息是固定的,有些信息直接来自平台上的某些 xml 配置文件,那些都属于 Configuration 原有的那一部分。而新扩充出来的那一部分,就是与具体作业相关的了,所以扩充以后称为 JobConf。

可以说,要在 Hadoop 上执行一个作业,用户需要做的事情主要有:

- (1)提供具体的 Mapper、Reducer,也许还有 Combiner 等程序模块(Java 类)。
- (2)填写作业单,在作业单中写明所使用的 Mapper、Reducer 等模块及其他有关信息。
- (3)提交作业单。

对于我们现在这个具体的作业,Mapper 和 Reducer 都来自体外,因为 ValueAggregatorJob 内部并未定义内嵌的 Mapper 和 Reducer。不过在这个示例中这二者同样也是由 Hadoop 提供的,就是同一目录中的源码文件 ValueAggregatorMapper.java 和 ValueAggregatorReducer.java。

现在我们正在做的是第二项。那么具体填写、设置了些什么呢?例如把 Mapper 设置成 ValueAggregatorMapper.class,把 Reducer 设置成 ValueAggregatorReducer.class。这个作业还安排了使用 Combiner,所以还要把 Combiner 设置成 ValueAggregatorCombiner.class。根据前述可知,这里 ValueAggregatorMapper.class 是对 ValueAggregatorMapper 类的描述,余可类推。从代码中可见,需要设置的内容还有不少,例如所需 mapper 和 reducer 的数量,Map 阶段的输出类型,最终的输出类型即整个 MapReduce 计算的输出类型,等等。

填写完这个作业单,程序就依次返回到 main(),用户的准备作业已完成,下一步就是通过 JobClient.runJob()提交作业了。我们的目的是要了解 Hadoop 内部的运转,因而需要顺着程序的流程深入进去。

这里对 runJob()的调用是直接通过类进行的,而不是通过这个类的具体对象进行的,程序中迄今尚未创建具体的 JobClient 类对象。其实都一样,总之是通过 JobClient 这个类的函数跳转表进行的。我们直接看 runJob()的代码。

```
[ValueAggregatorJob.main() > JobClient.runJob()]
```

```
public static RunningJob runJob(JobConf job) throws IOException {
    JobClient jc = new JobClient(job); //创建 JobClient 类的对象 jc
    RunningJob rj = jc.submitJob(job); //提交作业,返回一个 RunningJob 对象 rj
    try {
        if (!jc.monitorAndPrintJob(job, rj)){ //监视并显示作业 rj 的运行情况,直至作业结束
            throw new IOException("Job failed!"); //如果返回值是 false 就说明运行失败
        }
    } catch (InterruptedException ie) { //如果运行过程中 InterruptedException 异常
        Thread.currentThread().interrupt(); //就中断该线程的执行
    }
}
```

```

    }
    return rj;
}

```

这里创建具体的 `JobClient` 对象了,而作为参数传下来的 `JobConf` 类对象,即参数配置块 `job`,则用作创建 `JobClient` 对象的参数。`JobClient` 的体量较大,其结构大致如下:

```

class JobClient extends CLI{
    ] static final String MAPREDUCE_CLIENT_RETRY_POLICY_ENABLED_KEY =
        "mapreduce.jobclient.retry.policy.enabled"
    ] ... //还有好几个类似于这样的字符串
    ] static{ ConfigUtil.loadResources(); } //静态初始化
    ] UserGroupInformation clientUgi
        //UserGroupInformation 类的对象 clientUgi,用来表明客户身份
    ] public JobClient(JobConf conf) //本类对象的构造函数
        > init(conf) //创建 JobClient 对象时要调用其 init()方法
    ] public void init(JobConf conf)
        > setConf(conf)
        > cluster = new Cluster(conf)
        > clientUgi = UserGroupInformation.getCurrentUser() //获取客户的身份信息
    ] ... //跳过若干此刻不感兴趣的方法
    ] public synchronized FileSystem getFs()
        > return cluster.getFileSystem()
    ] public RunningJob submitJob(String jobFile) //提交作业,以文件名为参数
        > job = new JobConf(jobFile)
        > return submitJob(job)
    ] public RunningJob submitJob(final JobConf conf) //提交作业,以 JobConf 对象为参数
        > return submitJobInternal(conf) // submitJob()通过 submitJobInternal()完成作业提交
    ] public RunningJob submitJobInternal(final JobConf conf) //实际提交作业的过程
        > ... //详见下述
    ] ... //跳过更多此刻不感兴趣的方法
    ] public static RunningJob runJob(JobConf job) //提交并监视作业的运行,直至其结束
    ] public boolean monitorAndPrintJob(JobConf conf, RunningJob job)
        > return ((NetworkedJob)job).monitorAndPrintJob()
    ] public void setTaskOutputFilter(TaskStatusFilter newValue)
    ] ...
    ] public static void main(String argv[])
        > res = ToolRunner.run(new JobClient(), argv)
        > System.exit(res)
    ] static class NetworkedJob implements RunningJob{ //JobClient 内部定义的类
    ] Job job // NetworkedJob 里面有一个 Job 对象
}

```

```
[] public NetworkedJob(JobStatus status, Cluster cluster) //该类对象的构造函数
>[] public Configuration getConfiguration()
>[] public String getJobName()
>[] ... // NetworkedJob 类中还有些我们暂时不感兴趣的方法
```

这里要对上述内容做一些解释。

首先, JobClient 这个类是对另一个类 CLI 的扩展, 而 CLI 是“Command Line Interface”即“命令行界面”的缩写, 这个类将来我们还会碰到。所以, JobClient 实质上是个经过扩充的 Hadoop 命令行使用界面。这个类有自己的 main() 函数, 因而是可以直接通过命令行(例如“java JobClient …”)启动作为一个 JVM 进程运行的; 但是如果通过 new 加以创建, 那程序就不走它的 main(), 而走它的构造函数 JobClient(), 那就不另起一个 Java 虚拟机了。

如前所述, “类”(以及由类的实体化而来的“对象”)既包含数据, 又包含用来处理这些数据的数据程序即“方法”, 每一项数据可以是原始类型的数据, 如整数、浮点数、布尔量等, 也可以是某个类的对象。例如这里就有个 String 类的对象 MAPREDUCE_CLIENT_RETRY_POLICY_ENABLED_KEY, 注意这是个作为 String 对象的变量名, 而不是像 C 程序中所使用的宏定义。这个变量的值是“mapreduce.jobclient.retry.policy.enabled”, 而且是 final, 即不可更改的, 实际上这是用来查询配置文件的, 平台上的某个.xml 配置文件中应该有以此为键名的值, 要是更改了就查询不到了。此外, 这个对象的定义前面有个 static 标志, 说明只存在于类的 Class 数据结构中, 而不存在于具体对象的数据结构中。换言之, 同属 JobClient 类的所有对象都共享此项数据。

然后, 这里有个形如 static{ … } 的无名方法, 称为“静态初始化”方法, 用于整个类的初始化。当 Java 虚拟机按 import 语句的指示装载 JobClient 这个类的时候, 要执行这个静态初始化方法, 这里面只有一个语句——ConfigUtil.loadResources(), 那就是执行 ConfigUtil 类所提供的方法 loadResources(), 以装载相关的资源, 实际上就是一些采用 XML 语言的配置文件, 包括 mapred-default.xml、mapred-site.xml、yarn-default.xml、yarn-site.xml。

JobClient 中还内嵌定义了一个成员类 NetworkedJob, 这是为远程管理而设置的, 我们在这里不感兴趣。

接着是一个 UserGroupInformation 类的对象 clientUgi。这个成分对于用户访问权限, 从而对于系统的安全性起着关键的作用, 本书后面还要详述。顺便说明, 像 clientUgi 这样的变量形式的成分, 并不代表着整个 UserGroupInformation 类对象的实体, 而只是个“指引(Reference)”, 实质上就是个结构指针, 所以这里只是为这个指针分配内存空间。较之 C 语言中的结构指针, Java 语言中 Reference 的特殊之处在于: 凡有给这个指针赋值的语句时, 编译器(或解释器)会检查所赋之值是否确实指向一个该类对象, 并且不允许在 Java 程序中对此进行运算, 所以比 C 语言中的指针安全。

除上面所述的这些以外, 剩下的就是 JobClient 类所提供的种种方法函数了。注意, 这里所列的各种方法其实都可能多个版本, 方法名相同但参数表不同, 这就是多态。

那么, 在我们这个情景中, 通过 new 操作创建 JobClient 对象时在其构造方法中都做了些什么呢? 从上面的摘要中可以看出, 执行的是 JobClient 类所提供的方法 init()。而这个 init(), 则做了三件事情。一是 setConf(conf), 将创建对象时传下的 conf 设置成为本对象的配置块, 也就是从其创建者那里继承了它的配置块。二是创建一个 Cluster 类的对象 cluster, 这是对于运行着 Hadoop 的计算机集群的一个笼统的描述, 实际上只是对于如何与外界联系的

描述,其内容主要来自配置块,而并不涉及集群中那些具体的节点。注意这里被赋值的变量 `cluster`,我们在 `JobClient` 类的定义中找不到这个变量,当然它也不可能是全局变量,因为 Java 语言不允许有全局变量。实际上,这个变量是从 `JobClient` 的父类 `CLI` 那里继承下来的,`JobClient` 是对 `CLI` 类的扩充。再看第三件事,那就是通过 `UserGroupInformation` 类的方法 `getCurrentUser()` 获取当前用户的身份信息,那是一个 `UserGroupInformation` 类(缩写成 `UGI`)的对象。前面我们已经看到 `JobClient` 内部有个该类对象 `clientUgi`,但是并未见到赋值,所以只是一个空白的对象指引,相当于一个未经初始化的变量,现在经过 `getCurrentUser()` 才有了相关的信息。

做完这些事情之后,前面 `runJob()` 中创建 `JobClient` 对象 `jc` 这一步就完成了。

接着是调用 `jc.submitJob(job)`,就是调用 `JobClient` 类的方法 `submitJob()`。从前面的摘要中可见,这个方法所做的事情就只是调用 `submitJobInternal(conf)`,那才是关键所在。

由于 `submitJobInternal()` 这个方法相对比较复杂,我们在摘要中没有加以展开,需要专门加以分析。当然,这个方法也是由 `JobClient` 提供的。

```
[ValueAggregatorJob.main() > JobClient.runJob()
> JobClient.submitJob > JobClient.submitJobInternal()]

public RunningJob submitJobInternal(final JobConf conf) ... {
    try {
        conf.setBooleanIfUnset("mapred.mapper.new-api", false); //mapper,老 API
        conf.setBooleanIfUnset("mapred.reducer.new-api", false); //reducer,老 API
        Job job = clientUgi.doAs (new PrivilegedExceptionAction<Job> () {
            //以实际用户的权限行事

            @Override //表示下面这个 run() 是对于从父类继承的同名方法函数的覆盖
            public Job run() throws IOException, ClassNotFoundException, InterruptedException {
                Job job = Job.getInstance(conf);
                job.submit(); //以实际用户的权限,通过 Job.submit() 提交作业
                return job;
            }
        }); // clientUgi.doAs() 结束于此,回到原来的身份和权限
        // update our Cluster instance with the one created by Job for submission
        // (we can't pass our Cluster instance to Job, since Job wraps the config
        // instance, and the two configs would then diverge)
        cluster = job.getCluster();
        return new NetworkedJob(job); //创建一个 NetworkedJob 对象,以便了解作业的进展
    } catch (InterruptedException ie) {
        throw new IOException("interrupted", ie);
    }
}
```

在软件运行的过程中,用户的身份是根据其属于哪一个“组(group)”来判定的,这种身份

决定了具体的用户有些什么权限。在单机上这个问题相对比较简单,然而在跨机器节点的“服务/客户”结构的系统中就比较复杂了。作为服务端的进程很可能是由系统管理员启动的,因而具有“超级用户”的权限,但是做的事情、提供的服务,却可能是为普通用户的,需要以普通用户的名义去做。打个比方,我们去某个政府部门办事,办事的人是公务员,当然有比较高的权限,例如可以查看许多信息,可是如果要把这些信息发送给申请人,那就不可以了,因为申请人按规定没有那样高的权限。所以,办事的人必须区分,哪些事是以公务员的身份在做,哪些事是以申请人的名义、按申请人的身份在做。通常,以申请人的身份意味着权限的降低。

但是也有倒过来的,有时候会碰到一些需要更高权限才能完成的事情,这时候需要暂时赋予用户更高的权限,使其得以完成相关的操作。当然,这个更高的权限也不是随便就给、谁都给的,通常会在屏幕上弹出一个窗口要求用户输入密码,核对密码无误后才能往下走,到完成了需要更高权限的操作之后,就又降低到用户原先的权限。Unix/Linux 的“su”机制就是这样。

Java 语言 (Java 虚拟机) 也提供了这么一种机制。这里的 `clientUgi.doAs()`, 即 `UserGroupInformation.doAs()` 就属于此种机制,意为以谁谁谁的身份做什么什么。以谁的身份呢? 以具体 `UserGroupInformation` 对象所述的那个组的身份。以这个身份做什么呢? 做调用参数 `PrivilegedExceptionAction` 所定义的行动。这是需要严格查验和控制操作权限的行动(如果启用了安全模式的话)。可是这个行动究竟是什么呢? 那就是后面花括号里面的那个 `run()` 函数。实际上 `PrivilegedExceptionAction` 是 JDK 中定义的一个界面,凡是实现这个界面的类,都必须提供一个 `run()` 函数,供 `doAs()` 调用。对于类的定义和创建,通常我们采用静态定义,那就是通过例如“`class MyAction implements PrivilegedExceptionAction { ... }`”这样的形式加以定义,然后就通过“`new MyAction(...)`”加以创建。但是也可以采用像这里的动态定义,把定义和创建都放在一个复合语句中。这里的“`new PrivilegedExceptionAction`”表示创建一个实现了 `PrivilegedExceptionAction` 界面的某个无名类对象,而后面的花括号中就是这个类的定义。当然,动态定义只适用于形式上比较简单的类,像这里只有一个方法,就是 `run()`; 复杂一些的类就不适合动态定义了,因为那会使程序的可读性变差。

另外,界面 `PrivilegedExceptionAction<Job>` 的定义是一种模板,说明像这样同名的界面可以有好多,其中之一是针对 `Job` 类的。这样,“`new PrivilegedExceptionAction<Job>()`”就是创建一个实现了针对 `Job` 类的 `PrivilegedExceptionAction` 界面的某类对象。

还有个问题,这里的 `clientUgi` 是怎么来的? 前面说了,创建 `JobClient` 对象时的 `init()` 阶段所做的事情之一就是通过对 `UserGroupInformation.getCurrentUser()` 获取当前用户的 UGI。但是这个 `getCurrentUser()` 又怎么能得到应该放在 UGI 中的那些信息呢? 在用户这一边,这些信息来自宿主操作系统。以 `ValueAggregatorJob` 这个作业为例,用户是在某个节点机的宿主操作系统上通过类似于“`java ValueAggregatorJob ...`”这样的命令行发起一个 JVM 进程,来执行这个类的 `main()` 函数,以提交作业的。在宿主操作系统上,这个 JVM 进程属于当前用户,而当前用户是通过 login 过程进入系统的,所以操作系统知道这个用户是谁,属于哪一个组。所以, `UserGroupInformation.getCurrentUser()` 通过操作系统可以获取当前用户的身份信息。

不仅如此,这里所创建的 UGI 将会伴随着这个作业的整个生命周期,不管这个作业被提交到哪里,指派到哪里,构成这作业的任务被分配到哪里,这个 UGI 都会如影相随。

回到所创建 PrivilegedExceptionAction 对象的 run() 函数:

```
[ValueAggregatorJob.main() > JobClient.runJob() > JobClient.submitJob >
JobClient.submitJobInternal() > UserGroupInformation.doAs() >
PrivilegedExceptionAction.run()]
```

```
public Job run(){
    Job job = Job.getInstance(conf);
    job.submit(); //提交作业
    return job;
}
```

这里先通过 Job 类的 getInstance() 方法获取代表着本作业的 Job 类对象, 实际上就是创建一个 Job 类的对象 job。注意用作 getInstance() 参数的 conf 是个 JobConf 类对象, 这是从上面一路传下来的, 有关本次作业的所有信息都在这个对象(数据结构)中。然后, 就调用这个 Job 类对象的 submit() 函数, 提交这个作业。

值得注意的是, 在 Hadoop 的代码中, 有两个关于 Job 类的定义, 一个在 mapred 分支上, 另一个在 mapreduce 分支上。而 submitJobInternal() 也有两个, 我们正在分析的是作为 JobClient 类的一个方法函数的 submitJobInternal(), 也在 mapred 分支上, 所以是“老”的作业提交方法。那么, 按说这里引用的 Job 类也应该是定义于 mapred 分支上的那一个了? 然而偏不是。要知道这里引用的是哪一个 Job 类, 最可靠的办法是看看源码文件头部的 import, 看它导入的究竟是哪一个 Job 类。事实上在 JobClient.java 文件头部有这么一行:

```
import org.apache.hadoop.mapreduce. Job;
```

所以这里引用的 Job 类是定义在 mapreduce 分支上的那一个。也就是说, 程序的流程转到新 API 上去了。

这又是一个体量挺大的类型, 定义了许多方法, 下面的定义摘要只列出了我们马上就要用到的成分, 其余的则在将来要用到的时候再加说明:

```
class Job extends JobContextImpl implements JobContext {}
] ... //数据部分, 还有更多的数据在父类 JobContextImpl 中
] static { ConfigUtil.loadResources(); } //本类的静态初始化
] Job(JobConf conf) //Job 类的构建方法
] > super(conf, null) //调用父类 JobContextImpl 的构建方法
] > this.credentials.mergeAll(this.ugi.getCredentials())
] getInstance(Configuration conf) //根据 Configuration 创建 Job 对象
] > jobConf = new JobConf(conf)
] > new Job(jobConf) //创建 Job 类对象
] getCluster()
] setUseNewAPI()
] connect()
] getJobSubmitter()
```



```

] submit()
  > ... //详见后述
] ... //更多的操作方法

```

这个 Job 类提供一个方法 `getInstance(Configuration conf)`, 它要求的调用参数是 Configuration 类对象, 而上面 `run()` 给出的 `conf` 是 JobConf 类的。不过这倒不要紧, 因为 JobConf 是对 Configuration 的扩充, 可以被用作 Configuration。可是 `getInstance()` 所要做的其实是创建一个 Job 类对象, 创建时所需的参数却又是 JobConf 类对象, 所以这里又根据 Configuration 类对象再创建起一个 JobConf 类对象。其实这里只要做一下类型转换 (cast), 或者在 Job 类中增加一个以 JobConf 为参数的 `getInstance()`, 就不需要这么来回绕了。至于 Job 类对象的创建, 我们这里就不追下去了。

回到前面 `run()` 的代码中, 现在已经根据上面传下来的 JobConf 创建了一个 Job 对象, 下一步就是调用它的方法 `submit()`, 即 `Job.submit()`, 进而完成作业的提交。

而 `Job.submit()`, 则正是前面所说三种作业提交方法的汇聚点。从调用 `Job.submit()` 开始, 在“地方”上的流程就进入了第二阶段。

5.3 示例二: 采用新 API 的 WordCount

示例一采用的是老 API, 现在已经很少用了。Hadoop 从 1.0 版之后就把重点转到了新 API, 现在只是为了保持兼容性才保留了老 API。所以, 现在 Hadoop 的代码中已经难以找到采用老 API 的示例了, 而采用新 API 的示例则有很多。本节采用的示例是个比较典型的 MapReduce 应用 WordCount, 用来统计目标文件中的词频, 就是不同的单词各出现了多少次, 其源码文件在 `hadoop-mapreduce-project/hadoop-mapreduce-examples/src` 目录下面, 相对路径名为 `main/java/org/apache/hadoop/examples/WordCount.java`。

作为一个可 (在 Java 虚拟机 JVM 上) 独立执行的“可执行程序”, WordCount 也是个带有 `main()` 方法的 Java 类。我们先看一下它的摘要:

```

class WordCount{
] class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{}
]] map(Object key, Text value, Context context)
] class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable>{}
]] reduce(Text key, Iterable<IntWritable> values, Context context)
] main(String[] args)

```

这可以说是个最小、最简单, 却又很典型的 MapReduce 应用示例。WordCount 这个类中内嵌定义了两个类: TokenizerMapper 和 IntSumReducer。前者是对 Mapper 类的扩充, 所以实质上就是一个 Mapper, 但是其中的 `map()` 方法是自己提供的; 后者则是对 Reducer 类的扩充, 所以本质上是个 Reducer, 但是其中的 `reduce()` 方法是自己提供的。这就是用户程序员要提交给 Hadoop 的自备 Mapper 和 Reducer。除这二者之外, 就只有一个 `main()` 函数了。

注意, TokenizerMapper 所继承和扩充的类是 Mapper, 那是 Hadoop 提供的默认 Mapper。如果你不提供 Mapper, 那么 Hadoop 就给你用上它所提供的默认 Mapper。Mapper 这个类是

以模板的形式定义的：

```
class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>{}
```

就是说的在扩充或直接采用这个类的时候应该把抽象参数类型 KEYIN、VALUEIN 等根据实际需要落实为具体的类型。这里 KEYIN 和 VALUEIN 是 Mapper 的输入 KV 对 (Key/Value 对) 的类型；KEYOUT 和 VALUEOUT 则是其输出 KV 对的类型。这样，编译以后可以因具体的输入/输出 KV 对类型而生成各种不同的 Mapper，这些 Mapper 的参数类型各不相同，但是程序的结构和流程都一样。在这里的 WordCount 中，KEYIN 的类型落实为 Object，实际上就是什么类型都可以 (除无结构的原始类型如 Int 等之外)，因为在 Java 语言中所有的类都是从 Object 直接或间接继承扩充而来的；而 VALUEIN 则落实为 Text。这个意思是：对于 Mapper 的输入 KV 对，任何类型都可以定义为 Key 的类型，但是 Value 的类型则一定是 Text。

Mapper 是这样，Reducer 也是一样的道理，只不过 Reducer 的输入 KV 对的类型是 Text/IntWritable。

由于本章的重点在于作业的提交过程，我们先把 WordCount 的 Mapper 和 Reducer 放一下，留待后面讲解 Hadoop 的 MapReduce 框架时再回头细究。现在我们把目光聚焦在它的 main() 方法。

```
[WordCount.main()]
```

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
        System.err.println("Usage: wordcount <in> [<in>...] <out>");
        System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++ i) {
        FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
        new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1); //提交作业并等待其结束运行
}
```

显然,除没有使用 Combiner 之外,这里做的事情跟示例一中所做的很相似,也是先准备好一份类似于作业单那样的东西,然后就把所需的材料,包括 Mapper 和 Reducer,随同作业单一起提交给 Hadoop。所不同的是,在采用老 API 的示例一中,作业单是个 JobConf 类的对象,提交作业时就调用 JobClient.runJob()。而现在,在新 API 中,所用的作业单改成了一个 Job 类对象,要提交作业时就调用其 waitForCompletion()方法,这个调用要到作业结束运行后才返回。这里的最后一行代码表示:对于 Job.waitForCompletion()的调用,如果返回 true 就以 0 为参数调用 System.exit(),以退出程序运行并让操作系统上的 Shell 知道这是正常结束;否则就以 1 为参数,让 Shell 知道运行失败。这是规矩,对于 Shell 脚本的使用很重要。

从表面上看,新老 API 之间的差别似乎微不足道,但如果深入了解就发现不是那么回事了。前面讲过,TokenizerMapper 继承和扩充了 Mapper 类,所以本质上就是一个 Mapper;IntSumReducer 则继承和扩充了 Reducer 类。因为 Mapper 本来就是类,就可以直接被用作默认的 Mapper;Reducer 也是一样。可是如果我们仔细看一下示例一中的 ValueAggregatorMapper 和 ValueAggregatorReducer,则可以发现这二者都是继承和扩展了 ValueAggregatorJobBase,而后者则实现了 Mapper 和 Reducer 两个界面。也就是说,在老 API 中 Mapper 和 Reducer 不是类而只是界面。这就是一个明显的不同,当然还有别的不同。

不过,如前所述,示例一中采用老 API 的作业提交,结果也走到了新 API 的 Job 类这一边,汇聚到 Job.submit()。但是早期的 Hadoop 不是这样,这是有了新 API 以后才把老 API 嫁接过来的。因为这属于 API 的内部实现,所以这样的改变并不影响用户。

正因为老 API 已被嫁接到了新 API 上,我们在示例一中已经看到过 Job 类的一部分摘要,但是那时候我们还不关心 waitForCompletion()和与此有关的内容,现在需要看看这个类的更详细的内容摘要了。

```
class Job extends JobContextImpl implements JobContext {}
] public static enum JobState {DEFINE, RUNNING}
] static final long MAX_JOBSTATUS_AGE = 1000 * 2
] static final String OUTPUT_FILTER = "mapreduce.client.output.filter"
] static final String COMPLETION_POLL_INTERVAL_KEY =
                                "mapreduce.client.completion.pollinterval"
] ...
] static final int DEFAULT_MONITOR_POLL_INTERVAL = 1000
] static final int DEFAULT_TASKLOG_TIMEOUT = 60000
] ...
] boolean waitForCompletion(boolean verbose)
] submit()
] setSpeculativeExecution()
] setInputFormatClass()
] setOutputFormatClass()
] setOutputKeyClass()
] setOutputValueClass()
] setMapOutputKeyClass()
```

```

] setMapOutputValueClass()
] setMapperClass()
] setCombinerClass()
] setReducerClass()
] getConfiguration()
] monitorAndPrintJob()
] printTaskEvents(TaskCompletionEvent[] events, ...)

```

首先,在 Job 这个类中有不少静态的数据和类型定义。比方说这里定义了一个枚举类型 JobState,其数值可以是 DEFINE 或 RUNNING,分别表示作业当前是处于定义/准备阶段还是运行阶段。而常数 MAX_JOBSTATUS_AGE 设置成 2000,表示最多 2000 毫秒就要刷新一下这个作业的状态。字符串常数 OUTPUT_FILTER 表明 MapReduce 的输出是否还需要有一层过滤,如果需要的话是什么过滤器,这是要以“mapreduce.client.output.filter”为键名去 xml 配置文件中查询的。

我们最关心的当然是方法 waitForCompletion()的实现,下面是这个方法的代码。

```
[WordCount.main() > Job.waitForCompletion()]
```

```

public boolean waitForCompletion(boolean verbose)
    throws IOException, InterruptedException, ClassNotFoundException {
    if (state == JobState.DEFINE) { //确保该作业只提交一次,提交后即变成 RUNNING
        submit(); //通过 Job.submit()走完作业提交的流程
    }
    //提交之后就监视其运行,直至结束
    if (verbose) {
        monitorAndPrintJob(); //饶舌模式,周期性地报告作业的进展
    } else {
        //要不然,就周期性地询问作业是否已完成
        // get the completion poll interval from the client.
        int completionPollIntervalMillis = Job.getCompletionPollInterval(cluster.getConf());
        while (!isComplete()) { //询问作业是否已经完成
            try {
                Thread.sleep(completionPollIntervalMillis); //尚未完成,睡一会儿
            } catch (InterruptedException ie) { //如果因中断而被唤醒,就继续下一轮循环
            }
        } //end while
    }
    return isSuccessful();
}

```

这个方法,在其执行过程中有三种发生异常的可能,即 IOException、InterruptedException、ClassNotFoundException。因为作业最终得要提交到中央节点,这就涉及节点之间的通信,如果通信出了问题而且不能恢复,那么作业就无法提交,更谈不上执行了,这时候 JVM 就会对

所在的进程发起一次 `IOException` 异常, 让其退出运行。另一方面, 如果在执行过程中被别的线程或 JVM 中断, 那也无法继续执行了。但是, 这里有个例外, 就是在下面 `while` 循环中的 `try` 块在睡眠期间发生中断对作业的执行没有什么影响。因为此时当前进程已经完成作业提交, 只需过一会儿去了解情况, 作业的实际执行已经不在这个进程上, 即使发生中断也不影响作业的执行, 所以这里要拦截捕捉 (`catch`) 中断异常并将其丢弃 (空语句), 不让它落入覆盖着整个方法的异常处理机制。最后, `ClassNotFoundException` 发生于运行时要动态引用某个类的对象但是却找不到的情况下, 那当然也无法再往下执行了。

从作业提交流程的角度看, 这个方法的代码再简单不过了, 实际就是对 `Job.submit()` 的调用, 只是在调用之前要检查一下本作业是否处于 `DEFINE` 状态, 以确保一个作业不会被提交多次。如上所述, `JobState` 的值只有 `DEFINE` 和 `RUNNING` 两种, 具体 `Job` 对象创建之初在构造函数 `Job()` 中将其设置成 `DEFINE`, 作业提交成功之后就将其改成 `RUNNING`, 这就把门关上了。

在正常的情况下, `Job.submit()` 很快就会返回, 因为这个方法的作用只是把作业提交上去, 而无须等待作业的执行和完成。但是, 在 `Job.submit()` 返回之后, `Job.waitForCompletion()` 则要等待作业执行完成了以后才会返回。在等待期间, 如果参数 `verbose` 为 `true`, 就要周期地报告作业执行的进展, 或者就只是周期地检测作业是否已经完成。

读者也许会问, 代码中对 `submit()` 并不检查其返回值, 那万一在进一步提交作业的过程中出错呢? 其实这就是前述三种异常处理的作用所在, 因为要出问题也无非就是通信出错、找不到需要动态引用的类以及被强迫中断这么几种, 执行中没有发生异常就是成功。

下面就是 `Job.submit()` 的事了, 这就跟前面通过老 API 界面 `JobClient.runJob()` 提交作业的流程汇合了。

5.4 示例三: 采用 ToolRunner 的 QuasiMonteCarlo

除新、老两种 API 外, Hadoop 还提供了另一种开发 Hadoop 应用的方法和提交作业的途径, 那就是把应用做成对 Tool 界面的实现, 再借助 Hadoop 的 `ToolRunner` 来运行这个实现了 Tool 界面的对象。我们这里要介绍的示例三 `QuasiMonteCarlo` 就采用这种方法, 其源码文件在 `hadoop-mapreduce-project/hadoop-mapreduce-examples/src` 目录下面, 相对路径名为 `main/java/org/apache/hadoop/examples/QuasiMonteCarlo.java`。

`QuasiMonteCarlo` 这个应用采用准蒙特卡罗方法计算圆周率 π , 即 Pi 。其思路是这样的: 在单位正方形里以 $[0.5, 0.5]$ 为圆心、以 0.5 为半径作内切圆, 则圆的面积为 0.25π , 即 $\pi/4$ 。这样, 只要能得出这个内切圆的面积, 就知道 π 是什么数值了。那么, 怎样才能算得这个内切圆的面积呢? 我们可以把这个正方形打上很多网格, 然后统计所有的格点, 看有多少落在圆内 (包括圆周上), 有多少落在圆外, 二者之和就是格点总数。因为格点总数代表着单位正方形的面积, 即 1 , 落在圆内的格点数与总数之比就是圆的面积。如果精度不够, 就把网格打得再细一些。至于怎样判断一个格点是否落在圆内或圆周上, 那是很简单的, 因为根据每个格点的坐标可以算得其与圆心的距离, 如果大于 0.5 就是落在圆外。

对于 π 的一次计算, 网格一旦确定之后就不再变动, 对每一个格点的计算都是独立的, 并不依赖于其他格点的计算, 理论上所有格点的计算都可以并行, 但由于格点的数量很大 (不然

精度不够),实际上只能按块划分。可见,这是一种典型的“大数据小计算”的算法,这样的算法很适合 MapReduce。不采用此类算法的话, π 的计算是很难并行化的。

显然,这种方法是在拼计算能力,以计算能力换取算法的简化和并行化;某种意义上这也正是大数据处理的本质所在。

鉴于格点的数量很大,每个格点的计算都要并行是不现实的,所以实际上我们总是把这单位正方形划分成若干条块,每一条块中仍有很多个格点,把每一条块的格点计算交给一个 Mapper。这样,条块之间是并行的,但是条块之内仍是串行的,至于分多少个条块(需要多少个 Mapper)就视实际情况而定了。

先看一下 QuasiMonteCarlo 这个类的结构摘要:

```
class QuasiMonteCarlo extends Configured implements Tool {}
] class HaltonSequence{} // 二维 Halton Sequence 是用来生成取样点的算法
] class QmcMapper extends Mapper<...> {}
]] map(LongWritable offset, LongWritable size, Context context)
] class QmcReducer extends Reducer<..., WritableComparable<?>, Writable> {}
]] reduce(BooleanWritable isInside, Iterable<LongWritable> values, Context context)
] estimatePi(int numMaps, long numPoints, Path tmpDir, Configuration conf)
] run(String[] args)
] main(String[] argv)
> System.exit(ToolRunner.run(null, new QuasiMonteCarlo(), argv))
```

这个 Java 类有 main()方法,所以它本身是个独立的 JVM 可执行程序。此外,这个类一方面是对 Configured 类的扩充;另一方面又实现了 Tool 界面。那么 Tool 界面是什么样的呢?其实这是个很简单的界面:

```
public interface Tool extends Configurable {
    int run(String[] args) throws Exception;
}
```

这是对界面 Configurable 的扩展,在 Configurable 的基础上增加了一个方法 run()。而 Configurable 也是一个很简单的界面:

```
public interface Configurable {
    /* * Set the configuration to be used by this object. */
    void setConf(Configuration conf);
    /* * Return the configuration used by this object. */
    Configuration getConf();
}
```

所以,界面 Tool 只定义了三个方法,即 run()、setConf()和 getConf()。其中 setConf()和 getConf()已由 Configurable 类实现,QuasiMonteCarlo 类则补上了 run()的实现。后面我们会看到,只要把实现了 Tool 界面的类交给 ToolRunner.run(),后者就会为其创建一个对象并调用其 run()方法,其余就是这个 run()方法里面的事了。QuasiMonteCarlo 的其他方法都是用户自己定义的。

从 QuasiMonteCarlo 的摘要可以看出,这个类里面有内嵌的 QmcMapper 和 QmcReducer,二者分别是对 Mapper 和 Reducer 的扩充。由此可见,实际上采用的也是新 API,而实现着 Tool 界面的 QuasiMonteCarlo 类在某种意义上是对新 API 的包装。不过,这并不意味着 Tool 界面跟新 API 有什么内在的联系,实现 Tool 界面的类也可以采用老 API,那都是 run() 以内的事。

此外还有个内嵌的类 HaltonSequence,这跟 MapReduce 框架没什么关系,只是应用层上由用户定义的一个类,是实现前述准蒙特卡罗方法之所需。

除了这几个内嵌的类和方法 run()、main() 以外,还有个函数就是 estimatePi(),等一下我们会看到它的代码。

如上所述,实现 Tool 界面的类只提供一个 run() 方法,让 ToolRunner 调用就行了。所以 QuasiMonteCarlo 的 main() 方法很简单,就是创建一个 QuasiMonteCarlo 对象,把它连同命令行参数一起交给 ToolRunner,后者就会将其当成一个命令行来执行。

```
[QuasiMonteCarlo.main()]
```

```
public static void main(String[] argv) throws Exception {
    System.exit(ToolRunner.run(null, new QuasiMonteCarlo(), argv));
}
```

注意,通过 new 操作创建一个对象,例如 QuasiMonteCarlo 类的对象时,执行的是这个类的构造方法 QuasiMonteCarlo(),与其 main() 方法无关;而在启动一个 JVM 时,作为目标可执行类加载以后执行的则是其 main() 方法,但却并不直接创建这个类。

在这里,在 QuasiMonteCarlo 的 main() 方法中,则通过 new 操作创建一个 QuasiMonteCarlo 对象,并将其作为参数之一来调用 ToolRunner.run(),成为其调用参数 tool。我们看一下 ToolRunner 的摘要,ToolRunner 也是由 Hadoop 定义和提供的:

```
class ToolRunner {}
] run(Configuration conf, Tool tool, String[] args)
    > parser = new GenericOptionsParser(conf, args) //创建一个可选项解析器
    > tool.setConf(conf)
    > String[] toolArgs = parser.getRemainingArgs() //获取命令行中对于所启用工具的参数
    > tool.run(toolArgs) //启动工具的运行
] printGenericCommandUsage(PrintStream out)
    > GenericOptionsParser.printGenericCommandUsage(out)
] confirmPrompt(String prompt)
```

可见 ToolRunner 的核心就是 run()。运行什么呢?那就是实现了 Tool 界面的某个类的对象。除此之外,还有个方法是 printGenericCommandUsage(),那只是在人机界面上显示一段怎么使用命令行的提示。另一个方法是 confirmPrompt(),用来在屏幕上显示一个提示,然后要求用户输入 Y 或 N。所以,ToolRunner 就是用来帮助运行 Tool 的,是帮助我们运用“工具”的工具。

从 ToolRunner 的摘要可见,它的 run() 方法也很简单,说到底就是调用想要运行的 Tool

的 `run()` 方法,只是加上了对命令行的解析处理而已。读者或许要问:既然如此,那我在 `QuasiMonteCarlo.main()` 中直接调用 `run()`,而不到 `ToolRunner` 来这么绕一下,是否可以呢?其实也无妨,`ToolRunner` 只是为你提供帮助,让你的编程方便一些而已。

这样,我们的流程就到了 `QuasiMonteCarlo.run()`:

```
[QuasiMonteCarlo.main() > ToolRunner.run() > QuasiMonteCarlo.run()]
```

```
public int run(String[] args) throws Exception {
    if (args.length != 2) { //应该有两个命令行参数,即 nMaps 和 nSamples
        System.err.println("Usage: " + getClass().getName() + " <nMaps> <nSamples>");
        ToolRunner.printGenericCommandUsage(System.err); // ToolRunner 提供的便利
        return 2; //出错返回
    }

    final int nMaps = Integer.parseInt(args[0]);
    final long nSamples = Long.parseLong(args[1]);
    long now = System.currentTimeMillis();
    int rand = new Random().nextInt(Integer.MAX_VALUE); //产生一个随机数
    final Path tmpDir = new Path(TMP_DIR_PREFIX + "_" + now + "_" + rand); //临时目录

    System.out.println("Number of Maps = " + nMaps);
    System.out.println("Samples per Map = " + nSamples);
    //计算 Pi 并显示
    System.out.println("Estimated value of Pi is "
        + estimatePi(nMaps, nSamples, tmpDir, getConf()));

    return 0; //正常返回
}
```

这也是很简单的程序,把代码在这里列出只是为了让读者对此类程序有点直观的印象。显然,这里的核心在于对方法 `estimatePi()` 的调用。如前所述,对 π 值的估算是典型的大数据小计算,是最适合 MapReduce 并行计算的,这当然要提交给 YARN。

```
[QuasiMonteCarlo.main() > ToolRunner.run() > QuasiMonteCarlo.run() > estimatePi()]
```

```
public static BigDecimal estimatePi(int numMaps, long numPoints, Path tmpDir,
    Configuration conf) throws IOException, ClassNotFoundException, InterruptedException {
    //参数 numMaps 表示把正方形分成几块,实际上也是 Mapper 的个数
    //参数 numPoints 表示每块之中有多少点数
    Job job = new Job(conf);
    //setup job conf
    job.setJobName(QuasiMonteCarlo.class.getSimpleName());
    job.setJarByClass(QuasiMonteCarlo.class);
```

```

job.setInputFormatClass(SequenceFileInputFormat.class); //输入文件格式
job.setOutputKeyClass(BooleanWritable.class);
job.setOutputValueClass(LongWritable.class);
job.setOutputFormatClass(SequenceFileOutputFormat.class); //输出文件格式

job.setMapperClass(QmcMapper.class);
job.setReducerClass(QmcReducer.class);
job.setNumReduceTasks(1); //只要1个 reducer

// turn off speculative execution, because DFS doesn't handle
// multiple writers to the same file.
job.setSpeculativeExecution(false); //不要后备执行

//setup input/output directories
final Path inDir = new Path(tmpDir, "in");
final Path outDir = new Path(tmpDir, "out");
FileInputFormat.setInputPaths(job, inDir); //本作业的输入目录为"in"
FileOutputFormat.setOutputPath(job, outDir); //本作业的输出目录为"out"

final FileSystem fs = FileSystem.get(conf); //获取所使用的文件系统
if (fs.exists(tmpDir)) { //如果参数给定的临时目录已存在,有冲突
    throw new IOException("Tmp directory " + fs.makeQualified(tmpDir)
        + " already exists. Please remove it first."); //因发生 IOException 异常而终止运行
}
if (!fs.mkdirs(inDir)) { //如果创建输入目录失败
    throw new IOException("Cannot create input directory " + inDir); //异常,终止运行
}

try {
    //generate an input file for each map task
    for(int i = 0; i < numMaps; ++i) {
        final Path file = new Path(inDir, "part" + i); //输入目录中的文件名"part0"、"part1".....
        final LongWritable offset = new LongWritable(i * numPoints); //本块起点位移
        final LongWritable size = new LongWritable(numPoints); //本块大小(点数)
        final SequenceFile.Writer writer = SequenceFile.createWriter(fs, conf, file,
            LongWritable.class, LongWritable.class, CompressionType.NONE);
        try {
            writer.append(offset, size); //将位移和大小写入本块的输入文件
        } finally {
            writer.close(); //写完后就关闭本块的输入文件
        }
    }
}

```

```

    }

    System.out.println("Wrote input for Map #" + i);
} //end for

//start a map/reduce job
System.out.println("Starting Job");
final long startTime = System.currentTimeMillis();
job.waitForCompletion(true); //通过新 API 提交作业
final double duration = (System.currentTimeMillis() - startTime)/1000.0; //统计运行时间
System.out.println("Job Finished in " + duration + " seconds");

//read outputs
Path inFile = new Path(outDir, "reduce-out"); //输出目录中的文件名“reduce-out”
LongWritable numInside = new LongWritable(); //创建对象,用于落在圆内的点数
LongWritable numOutside = new LongWritable(); //创建对象,用于落在圆外的点数
SequenceFile.Reader reader = new SequenceFile.Reader(fs, inFile, conf); //打开文件
try {
    reader.next(numInside, numOutside); //从“reduce-out”读取落在圆内/圆外的点数
} finally {
    reader.close(); //读完后关闭文件
}

//compute estimated value
final BigDecimal numTotal //这是总的点数
    = BigDecimal.valueOf(numMaps).multiply(BigDecimal.valueOf(numPoints));
return BigDecimal.valueOf(4).setScale(20) //精度为小数点后 20 位
    .multiply(BigDecimal.valueOf(numInside.get()))
    .divide(numTotal, RoundingMode.HALF_UP); //计算圆内点数与总数的比率
} finally {
    fs.delete(tmpDir, true); //最后把临时目录删掉
}
}

```

代码中已添加了注释,再结合前面的说明,读者理解起来应该不会感到困难,就不再详细解释了。

这里调用 `job.waitForCompletion()` 之前的那一段就是我们已经熟悉的准备作业单的过程。MapReduce 的输出根据 `FileOutputFormat.setOutputPath()` 的设置写入目录“out”中的文件“reduce-out”。

同样,还是通过 `job.waitForCompletion()` 提交作业并监视其执行直到结束。当程序从 `job.waitForCompletion()` 返回时,MapReduce 计算已经完成,然后就是从文件“reduce-out”读出落入圆内的格点数量,算出 π 的数值。不过这里采用的不是一般的整数和浮点数,而是

LongWritable 和 BigDecimal, 因为一般整数和浮点数的容量和精度可能不够(这里设置的精度为小数点后 20 位)。

这里要简单提一下“后备执行(speculative execution)”。这是指 Hadoop 提供的一种机制,就是在根据用户要求安排一定数量的 Mapper 和 Reducer 的同时,还要准备好若干作为后备的 Mapper 和 Reducer,如果运行中发现某个 Mapper 或 Reducer 停滞不前或有别的反常现象,就安排后备的 Mapper 或 Reducer 顶上去,就像打仗时的“预备队”一样。但是,这里通过 `job.setSpeculativeExecution(false)` 否定了预备队的使用,注释中说这是因为 Hadoop 的分布式文件系统 DFS 不能处理多个线程同时写入同一文件。注意,这里 Reducer 的数量只是 1,但是后备 Reducer 顶上去的时候可能会有一小段时间与处于“一线”的 Reducer 重叠。

至于 QmcMapper 和 QmcReducer 的代码,我们就不深入进去了,读者可以自己看源文件。这里只是简单提一下。QmcMapper 的数量应该等于 numMaps,每个 Mapper 对应着一个输入文件,如 part0、part1 等,并从输入文件读入一个 offset 和一个 size。offset 决定了分块所在的位置,size 就是需要生成的点数。随后,每个 Mapper 都进入一个循环,依次调用 HaltonSequence 这个类所提供的方法,生成一个点,然后根据这个点的坐标计算其与圆心的距离,并对落在圆内和圆外的点分别计数。循环结束,即生成并计算了 size 个点之后,就把两个计数发送给 Reducer。而 Reducer 则接受来自所有 Mapper 发来的计数并加以汇总,并把结果写入输出文件。可见,在这个示例中 Reducer 的负担是很轻的。

总之,通过 ToolRunner 提交作业的方法,实质上与前面两种并无不同,只不过这样一来增加了一些灵活性,可以像别的 Tool 一样被启用而已。

从 `Job.waitForCompletion()` 往前一步,就是 `Job.submit()` 了。

5.5 从 Job.submit() 开始的第二段流程

如前所述,提交作业的三种方法都汇聚到了 `Job.submit()`。到了 `Job.submit()`,提交作业的流程就进入了“地方”上的第二个阶段,要把作业提交到“中央”去了。如果把 Hadoop 集群中的每一个节点都看作一个岛屿,那么这就是要“出海”,涉及跨节点的操作了。

我们在前面做过 Job 类的摘要,但是那其实只是一部分,现在我们需要从另一个视角再做一个摘要了:

```
class Job extends JobContextImpl implements JobContext {}
] submit()
] setUseNewAPI()
] connect()
] getJobSubmitter(FileSystem fs, ClientProtocol submitClient)
  > new JobSubmitter(fs, submitClient)
] isUber()           //是否“拼车”模式(MapTask 与 ReduceTask 在同一节点上)
] setPartitionerClass()//Mapper 的输出可能要由 Partitioner 按某种规则分发给多个 Reducer
] setMapSpeculativeExecution() //是否需要 Speculative 的 Mapper 起预备队的作用
] setReduceSpeculativeExecution() //是否需要 Speculative 的 Reducer 起预备队的作用
] setCacheFiles()
```

] ...

我们看 Job.submit() 的代码。

[Job.submit()]

```
public void submit() {
    ensureState(JobState.DEFINE); //确认没有重复提交
    setUseNewAPI();               //根据配置信息确定是否采用新 API
    connect();                     //建立与集群的连接,创建 Cluster 对象 cluster
    final JobSubmitter submitter =
        getJobSubmitter(cluster.getFileSystem(), cluster.getClient());
    status = ugi.doAs(new PrivilegedExceptionAction<JobStatus>() {
        public JobStatus run(){
            return submitter.submitJobInternal(Job.this, cluster);
        }
    });
    state = JobState.RUNNING;
    LOG.info("The url to track the job: " + getTrackingURL());
}
```

这里先说明一下 Job.setUseNewAPI()。这个函数根据配置文件中的若干配置项确定本作业所采用的是新 API 还是老 API,并生成显式的配置项“mapred.mapper.new-api”和“mapred.reducer.new-api”,并将之写入内存中的配置块,即 Job.conf 对象(这是个 org.apache.hadoop.mapred.JobConf 对象,见 JobContextImpl 类的定义)。如前所述,提交作业的 API 有新老之分,最后都汇聚到 Job.submit(),都已转换成新 API。这里的 Job 是定义于新 API 的 org.apache.hadoop.mapreduce.Job。但是这 API 不仅仅是作业提交的 API,同时也是 MapReduce 计算框架的 API。更具体地说,Mapper 和 Reducer 的 API 也有新老之分。现在是时候进一步加以说明了。

在早期的 Hadoop 中,Mapper 和 Reducer 都定义为 interface。当然,界面 Mapper 上定义了函数 map(),界面 Reducer 上定义了函数 reduce()。而具体的应用,则须实现这两个界面,提供实际的 mapper 和 reducer。以前述示例一的采用老 API 的 ValueAggregator 为例,则先定义一个抽象类 ValueAggregatorJobBase:

```
abstract class ValueAggregatorJobBase<K1 extends WritableComparable, V1 extends Writable>
    implements Mapper<K1, V1, Text, Text>, Reducer<Text, Text, Text, Text> {}
```

这个抽象类虽说是实现了 Mapper 和 Reducer 两个界面,实际上却并未提供 map()和 reduce()这两种操作,那是由扩充了这个 ValueAggregatorJobBase 的 ValueAggregatorMapper 和 ValueAggregatorReducer 提供的,例如:

```
class ValueAggregatorMapper<K1 extends WritableComparable, V1 extends Writable>
    extends ValueAggregatorJobBase<K1, V1> {}
```

] map()

ValueAggregatorReducer 也是一样。这就是采用老 API 的 mapper 和 reducer。而新 API 就不一样了。在新 API 上,Mapper 和 Reducer 都是 class:

```
class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {}
] map()
```

```
class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {}
] reduce()
```

而具体的应用,例如 WordCount,则有:

```
class WordCount {}
] static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{}
] static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable>{}
]
```

所以,在新老两种 API 中,连 mapper 和 reducer 的类型都是不一样的。尽管作业提交的流程都汇入了新 API 上的 Job.submit(),但是当然应该告诉将来的执行者,这个具体的应用所采取的是新 API 还是老 API。

这个信息通过 Job.setUseNewAPI()设置在 Job.conf 中,这个 Job 类是新 API 上的 Job 类,这是对 JobContextImpl 的扩充,其成分 conf 是从后者继承来的。注意不要把这 Job 类跟老 API 上的 Job 类混淆,那是对 ControlledJob 的扩充。

之所以会如此地弯弯绕绕,应该是出于版本更新的需要。具体的代码就留给读者自己阅读和研究了。再次提醒读者,遇有同名的类型定义时,一定要注意是在什么 package 中定义,而加以引用的代码所 import 的又是哪一个 package。

回到上面的代码,既然要跨节点操作,就得建立起对外的联系,上面对 connect()的调用就起着这个作用。

然后,就像要找一个懂得怎样跟“中央”打交道提交作业的办事员一样,要创建一个专门干这活儿的 JobSubmitter,它的 submitJobInternal()方法就是专做这件事情的。当程序从 submitJobInternal()返回的时候,作业的提交已经完成,所以就把作业的状态改成 RUNNING。至于 ugi.doAs()的作用,前面在示例一中已经介绍过了。

初看之下 Job.submit()似乎没几行程序,但实际上却是相当复杂的一个过程。我们先看 Job.connect()的代码。

```
[Job.submit() > Job.connect()]
```

```
private synchronized void connect()
    throws IOException, InterruptedException, ClassNotFoundException {
    if (cluster == null) { //如果 cluster 尚未创建
        cluster = ugi.doAs(new PrivilegedExceptionAction<Cluster>() {
            public Cluster run() {
                return new Cluster(getConfiguration());
            }
        });
    }
}
```

```

    }
}

```

可见 connect() 的作用就是保证节点上有个 Cluster 类对象, 如果还没有, 就创建一个。顾名思义, Cluster 类对象中应该存有与集群有关的信息, 知道如何与集群打交道, 其中也包括与“中央”打交道。但是这些信息从哪里来呢? 这不是来自集群, 因为此刻我们还不知道怎么跟集群打交道。所以这些信息只能来自配置文件。我们看 Cluster 类的摘要:

```

class Cluster{
] ClientProtocolProvider clientProtocolProvider //集群条件下为 YarnClientProtocolProvider
] ClientProtocol client //在集群条件下, 这是与外界通信的渠道和规则
] static ServiceLoader<ClientProtocolProvider> frameworkLoader =
    ServiceLoader.load(ClientProtocolProvider.class)
] ... //其他数据
] static { ConfigUtil.loadResources(); } //类的静态初始化
    > addDeprecatedKeys(); //为保持与老 API 兼容而设
    > Configuration.addDefaultResource("mapred-default.xml");
    > Configuration.addDefaultResource("mapred-site.xml");
    > Configuration.addDefaultResource("yarn-default.xml");
    > Configuration.addDefaultResource("yarn-site.xml");
] Cluster(Configuration conf) //构造方法 1
    > this(null, conf)
] Cluster(InetSocketAddress jobTrackAddr, Configuration conf) //构造方法 2, 给定主节点地址
    > this.conf = conf
    > this.ugi = UserGroupInformation.getCurrentUser()
    > initialize(jobTrackAddr, conf)
] initialize(InetSocketAddress jobTrackAddr, Configuration conf)
    > for (ClientProtocolProvider provider : frameworkLoader) {
    >> ClientProtocol clientProtocol = null
    >> if (jobTrackAddr == null) {
    >>> clientProtocol = provider.create(conf)
    >> } else {
    >>> clientProtocol = provider.create(jobTrackAddr, conf)
    >> }
    >> clientProtocolProvider = provider
    >> client = clientProtocol
    > } //end for
] getFileSystem() //获取集群的文件系统
] getClient()
> return client
] ...

```

这里有个静态初始化过程,在此过程中通过 `ConfigUtil.loadResources()` 装载了一批“资源”,具体都是.xml 配置文件,主要有 `mapred-default.xml`、`mapred-site.xml` 和 `yarn-default.xml`、`yarn-site.xml`。其中前两个配置文件用于老通信机制的实现,后两个用于新机制的实现。此外还有一些现在已不再使用,只是为保持与老版本兼容所需的资源。前面讲过,静态初始化是针对类的,而不是针对具体对象的,所以只执行一次。当然,这些配置文件都存在于本节点的磁盘上,其中例如 `mapred-default.xml` 是全局性的,所有节点上的这个文件应保持一致,而 `mapred-site.xml` 是只针对所在节点的,不同节点上的配置必然有所不同。

创建对象的时候要对有初始赋值的变量成分进行赋值,所以这里的 `frameworkLoader` 会得到赋值。这个变量的类型是 `ServiceLoader<ClientProtocolProvider>`,就是针对 `ClientProtocolProvider` 类的 `ServiceLoader`,而且这就是通过 `ServiceLoader.load()` 装载的。

这里首先要解释一下什么是 `ClientProtocolProvider` 和 `ClientProtocol`。用户向 RM 节点提交作业,是要 RM 为其安排运行,所以 RM 起着服务提供者的作用,而用户则处于客户的位置。既然如此,双方就得有个协议,对于双方怎么交互,乃至服务怎么提供,都得有个规定。在 Hadoop 的代码中,这所谓 Protocol 甚至被“上纲上线”到了计算框架的高度,连是否采用 YARN 框架也被纳入了这个范畴。实际上 `ClientProtocol` 就起着这样的作用,而 `ClientProtocolProvider` 顾名思义是 `ClientProtocol` 的提供者,起着有点像是 Factory 的作用。至于 `ServiceLoader<ClientProtocolProvider>`,那是用来装载 `ClientProtocolProvider` 的。

`ServiceLoader` 是由 JDK 提供的一个类,在源文件 `Cluster.java` 的头部有一行“`import java.util.ServiceLoader;`”说明运行时需要导入这个工具性质的类。在 OpenJDK 的代码中,这个类的摘要是这样的:

```
class ServiceLoader<S> implements Iterable<S> {}
] String PREFIX = "META-INF/services/" //关于这个 service 的信息在某个目录下面
] Class<S> service
] ClassLoader loader
] LinkedHashMap<String,S> providers
] Iterator<S> iterator()
] load(Class<S> service)
] parse(Class service, URL u) //URL 可以是文件路径
```

就是说,`ServiceLoader` 是针对某种服务 S 的,在这里是 `ClientProtocolProvider`。关于这种服务的“元信息”在 Hadoop 源码某个分支上的“`META-INF/services/`”目录下。对于 `ClientProtocolProvider` 应有名为 `org.apache.hadoop.mapreduce.protocol.ClientProtocolProvider` 的文件存在。事实上,在 Hadoop 的源码包中有两个这样的文件,一个是在 `mapreduce-project/mapreduce-client/hadoop-mapreduce-client-jobclient` 分支上,另一个是在 `mapreduce-project/mapreduce-client/hadoop-mapreduce-client-common` 分支上。前者的内容(除注释之外)是路径 `org.apache.hadoop.mapred.YarnClientProtocolProvider`;后者的内容则为 `org.apache.hadoop.mapred.LocalClientProtocolProvider`。

`ServiceLoader` 实现了 `Iterable` 界面,提供一个 `iterator()` 函数,因而可以用在 for 循环中。它还提供了一个 `load()` 方法,可以通过 `ClassLoader` 加载 Class。此外,它还提供解析文件内容的功能。

装载了作为 ServiceLoader 对象的 frameworkLoader,其 LinkedHashMap 中就有了上述的两个路径,这样就可以通过其 iterator()函数依次引用这两个路径了。

然后,在 Cluster 类的构造函数中就会调用其 initialize(),目的是要创建 ClientProtocolProvider 和 ClientProtocol。

但是 ClientProtocolProvider 是个抽象类,这意味着只有继承和扩充了这个抽象类的具体类才能被实体化成对象。Hadoop 的源码中一共只有两个类扩充和落实了这个抽象类,那就是 LocalClientProtocolProvider 和 YarnClientProtocolProvider。

```
class LocalClientProtocolProvider extends ClientProtocolProvider{}
class YarnClientProtocolProvider extends ClientProtocolProvider{}
```

可想而知,由这两种 ClientProtocolProvider 提供的 ClientProtocol 也是不一样的。事实上 ClientProtocol 是个界面,实现了这个界面的类也有两个,分别为 LocalJobRunner 和 YARNRunner。但是实际使用的只能是其中之一。

那么究竟应该是哪一个呢?可以通过配置项加以设定。

配置文件 mapred-default.xml 中有个配置项“mapreduce.framework.name”,用来设置所用的计算框架。配置项的值可以是 local,classic 或 yarn。其中 classic 是指 2.0 版以前老的框架,那时候还没有 YARN;而 yarn 当然是指在集群上运行 YARN 框架;至于 local,那是在单机上进行 MapRduce 计算,严格说来那就不是 YARN 框架了。当然,在单机上运行比在集群上运行要简单得多,所以这两种情况下所创建的 ClientProtocol 是不一样的,一个是 LocalJobRunner,另一个是 YARNRunner。至于 classic 则已经不用了。

```
<property>
  <name>mapreduce.framework.name</name>
  <value>local</value>
  <description>The runtime framework for executing MapReduce jobs.
                  Can be one of local, classic or yarn.
</description>
</property>
```

注意,这里设置的是 local。这是因为,人们把 Hadoop 下载过来,第一步总是先在单机上试试,稍微熟悉一下,然后才会跑到集群上去。所以,把 Hadoop 安装到集群上去的时候要把这个配置项改成 yarn。Hadoop 的文档 hadoop-mapreduce-project/INSTALL 中讲了这个问题:

Step 8) Modify mapred-site.xml to use yarn framework

```
<property>
  <name>mapreduce.framework.name</name>
  <value>yarn</value>
</property>
```

除来自配置文件以外,也可以在命令行中通过参数加以指定,例如“-jobconf mapreduce.framework.name=yarn”。

有了这些背景,我们再看 Cluster 的 Initialize()操作:

[Job.submit() > Job.connect() > Cluster.Cluster() > Cluster.Initialize()]

```
private void initialize(InetSocketAddress jobTrackAddr, Configuration conf) throws ... {
    synchronized (frameworkLoader) { //不允许多个线程同时进入此段代码,需要加锁
        for (ClientProtocolProvider provider : frameworkLoader) {
            LOG.debug("Trying ClientProtocolProvider : " + provider.getClass().getName());
            ClientProtocol clientProtocol = null;
            try { //试图创建 ClientProtocol,即 LocalJobRunner 或 YARNRunner,视配置而定
                if (jobTrackAddr == null) {
                    clientProtocol = provider.create(conf);
                } else {
                    clientProtocol = provider.create(jobTrackAddr, conf);
                }
                if (clientProtocol != null) { //已经创建成功
                    clientProtocolProvider = provider;
                    client = clientProtocol;
                    //已经创建了 ClientProtocol 对象,YARNRunner 或 LocalJobRunner
                    LOG.debug("Picked" + provider.getClass().getName()
                        + "as the ClientProtocolProvider");
                    break; //装载 provider 成功并创建 clientProtocol 成功,跳出 for 循环
                } else {
                    LOG.debug("Cannot pick " + provider.getClass().getName()
                        + "as the ClientProtocolProvider - returned null protocol");
                }
            } catch (Exception e) { //创建失败,记入日志
                LOG.info("Failed to use " + provider.getClass().getName()
                    + " due to error: " + e.getMessage());
            }
        } //end for //本轮循环中未能创建 ClientProtocol,继续下一轮 for 循环
    } //end synchronized

    if (null == clientProtocolProvider || null == client) { //未能创建所需的对象
        throw new IOException(
            "Cannot initialize Cluster. Please check your configuration for "
            + MRConfig.FRAMEWORK_NAME + " and the correspond server addresses.");
    }
}
```

这里的 for 循环,是基于前述 ServiceLoader 中 iterator()的循环。实际上也就是对两个

ClientProtocolProvider 的循环,目的是要通过 ClientProtocolProvider.create() 创建用户所要求的 ClientProtocol,也无非就是 LocalJobRunner 或 YARNRunner。只要有一次创建成功,循环就没有必要继续了,因为只能有一种选择;但是,如果两次都失败,程序就无法继续了,因为不知道怎样让 RM 提供计算服务。而能否成功创建,则取决于前述配置项的设置。不过 ClientProtocolProvider 是抽象类,实际上依次进行尝试的是 LocalClientProtocolProvider 和 YarnClientProtocolProvider。假定第一轮循环时进行尝试的是前者,那么:

```
[Job.submit() > Job.connect() > Cluster.Cluster() > Cluster.Initialize() >
LocalClientProtocolProvider.create()]
```

```
public ClientProtocol create(Configuration conf) throws IOException {
    String framework = conf.get(MRConfig.FRAMEWORK_NAME, MRConfig.LOCAL_FRAMEWORK_NAME);
    //从配置块中获取配置项“mapreduce.framework.name”,默认“local”
    if (!MRConfig.LOCAL_FRAMEWORK_NAME.equals(framework)) {
        return null; //如果不是“local”就失败
    }
    conf.setInt(JobContext.NUM_MAPS, 1); //既然是 local 就没有必要用多个 Mapper
    return new LocalJobRunner(conf); //实际创建 LocalJobRunner 对象
}
```

这里涉及的几个字符串常量定义于界面 MRConfig:

```
public static final String FRAMEWORK_NAME = "mapreduce.framework.name";
public static final String CLASSIC_FRAMEWORK_NAME = "classic";
public static final String YARN_FRAMEWORK_NAME = "yarn";
public static final String LOCAL_FRAMEWORK_NAME = "local";
```

如果配置项的值是“local”,或者干脆就没有这个配置项,那就创建 LocalJobRunner,并把作业单中的 NUM_MAPS 设置成 1;因为既然是 local,在单机上计算就没有必要使用多个 Mapper,那样并没有性能上的好处。创建了 LocalJobRunner,循环就结束了。

如果有这么个配置项,但所设置的值不是“local”,那就失败了,那就试试别的,应该就是 YarnClientProtocolProvider:

```
[Job.submit() > Job.connect() > Cluster.Cluster() > Cluster.Initialize() >
YarnClientProtocolProvider.create()]
```

```
public ClientProtocol create(Configuration conf) throws IOException {
    if (MRConfig.YARN_FRAMEWORK_NAME.equals(
        conf.get(MRConfig.FRAMEWORK_NAME))) {
        return new YARNRunner(conf); //实际创建 YARNRunner 对象
    }
    return null;
}
```


如果配置项的值是“yarn”,那就创建 YARNRunner。

我们关心的是 Hadoop 在集群上的运行,所以假定选用 YarnClientProtocolProvider,所创建的 ClientProtocol 是 YARNRunner。后面我们将看到这个 YARNRunner 的作用。

注意,YarnClientProtocolProvider 有两个 create(),上面代码中根据是否提供 jobTrackAddr 而分别加以调用,但是现在 Hadoop 代码中对这两个 create()的调用殊途同归,最后归结到同一个 create(),所以 jobTrackAddr 其实是不起作用的,这应该是老版本留下的残余。

另外,frameworkLoader 是 Cluster 类的静态成分,因而只是在 JVM 装载 Cluster 类的时候装载一次,但是 LocalJobRunner 或 YARNRunner 则是在 Cluster 类的 Initialize()中创建,那就是每创建一个 Cluster 对象时都会被执行一次。然而每一个作业的提交都是由一个独立的 JVM 进程完成的,都在一个独立的 JVM 上,所以实际上每个作业都会有自己的 frameworkLoader,当然也就有自己的 LocalJobRunner 或 YARNRunner。

回到前面 Job.submit()的代码,下一步是对 getJobSubmitter()的调用。这个函数创建一个 JobSubmitter 类对象,然后 Job.submit()就调用它的 submitJobInternal()方法,完成作业的提交。创建 JobSubmitter 对象时的两个参数就是调用 getJobSubmitter()时的两个参数,就是 cluster.getFileSystem()和 cluster.getClient()。其中 cluster.getClient()返回的就是 YARNRunner 或 LocalJobRunner;而 cluster.getFileSystem()的返回结果,对于 YARNRunner 是 RM 节点上文件系统的 URL,对于 LocalJobRunner 则是本节点上的一个相对路径为“mapred/system”的目录。

创建了 JobSubmitter 对象后,调用其 submitJobInternal()的两个参数也仍旧是这两个参数。

下面是 JobSubmitter 类的摘要:

```
class JobSubmitter{
] FileSystem jtFs
] ClientProtocol submitClient //来自 Job.submit(),在集群条件下是 YARNRunner
] String submitHostName
] String submitHostAddress
] JobSubmitter(FileSystem submitFs, ClientProtocol submitClient) //构建方法
    > this.submitClient = submitClient //将参数 submitClient 复制到 JobSubmitter 中
    //这是在 Job.submit()中通过 cluster.getClient()获取的,在集群条件下是 YARNRunner
    > this.jtFs = submitFs//jt 是 JobTracker 的缩写,2.0 版之后的 Hadoop 已改成 RM
] compareFs(FileSystem srcFs, FileSystem destFs) //比较两个文件系统是否相同
] getPathURI()
] checkSpecs()
] copyRemoteFiles()
] copyAndConfigureFiles()
] copyJar(Path originalJarPath, Path submitJarFile,short replication)
] addMRFrameworkToDistributedCache()
] submitJobInternal(Job job, Cluster cluster) //将作业提交给集群
] writeNewSplits(JobContext job, Path jobSubmitDir)
```

] ...

此刻我们关心的焦点就是这个类所提供的方法 `submitJobInternal()`, 因为 `Job.submit()` 就是通过这个方法将作业提交到集群的。这个方法的代码有点长, 我已对其稍作整理。

```
[Job.submit() > JobSubmitter.submitJobInternal()]
```

```
JobStatus submitJobInternal(Job job, Cluster cluster) throws ClassNotFoundException, ...{
    checkSpecs(job); //validate the jobs output specs, 检查输出格式等配置的合理性
    Configuration conf = job.getConfiguration();
    addMRFrameworkToDistributedCache(conf);
```

```
    Path jobStagingArea = JobSubmissionFiles.getStagingDir(cluster, conf); //获取目录路径
```

```
    //configure the command line options correctly on the submitting dfs
```

```
    InetAddress ip = InetAddress.getLocalHost(); //获取本节点(主机)的 IP 地址
```

```
    if (ip != null) {
```

```
        submitHostAddress = ip.getHostAddress(); //本节点 IP 地址的字符串形式
```

```
        submitHostName = ip.getHostName(); //本节点名称
```

```
        conf.set(MRJobConfig.JOB_SUBMITHOST, submitHostName);
```

```
        conf.set(MRJobConfig.JOB_SUBMITHOSTADDR, submitHostAddress);
```

```
    }
```

```
    JobID jobId = submitClient.getNewJobID(); //生成一个作业 ID 号
```

```
    job.setJobID(jobId); //将作业 ID 号写入 Job 对象
```

```
    Path submitJobDir = new Path(jobStagingArea, jobId.toString());
```

```
    //本作业的临时子目录名中包含着作业 ID 号码
```

```
    JobStatus status = null;
```

```
    try {
```

```
        conf.set(MRJobConfig.USER_NAME, //用户名
```

```
            UserGroupInformation.getCurrentUser().getShortUserName());
```

```
        conf.set("hadoop.http.filter.initializers", //准备用于 Http 接口的过滤器初始化
```

```
            "org.apache.hadoop.yarn.server.webproxy.amfilter.AmFilterInitializer");
```

```
        conf.set(MRJobConfig.MAPREDUCE_JOB_DIR, submitJobDir.toString());
```

```
        LOG.debug("Configuring job " + jobId + " with " + submitJobDir + " as the submit dir");
```

```
    // get delegation token for the dir /* 准备好与访问权限有关的证件(token) */
```

```
    TokenCache.obtainTokensForNamenodes(job.getCredentials(),
```

```
        new Path[] { submitJobDir }, conf); //获取与 NameNode 打交道所需证件
```

```
    populateTokenCache(conf, job.getCredentials());
```

```
    // generate a secret to authenticate shuffle transfers
```

```
    if (TokenCache.getShuffleSecretKey(job.getCredentials()) == null) {
```

```
        //需要生成 Mapper 与 Reducer 之间的数据流动所用的密码
```

```

KeyGenerator keyGen;
try {
    keyGen = KeyGenerator.getInstance(SHUFFLE_KEYGEN_ALGORITHM);
    keyGen.init(SHUFFLE_KEY_LENGTH);
} catch (NoSuchAlgorithmException e) {
    throw new IOException("Error generating shuffle secret key", e);
}
SecretKey shuffleKey = keyGen.generateKey();
TokenCache.setShuffleSecretKey(shuffleKey.getEncoded(), job.getCredentials());
} //end if

copyAndConfigureFiles(job, submitJobDir); //将可执行文件之类拷贝到 HDFS 中
Path submitJobFile = JobSubmissionFiles.getJobConfPath(submitJobDir); //配置文件路径

// Create the splits for the job /* 将输入数据文件切片,并写入临时目录 */
LOG.debug("Creating splits at " + jtFs.makeQualified(submitJobDir));
int maps = writeSplits(job, submitJobDir); //生成切片,以切片数量决定 Mapper 数量
conf.setInt(MRJobConfig.NUM_MAPS, maps); // "mapreduce.job.maps"
LOG.info("number of splits:" + maps);

// write "queue admins of the queue to which job is being submitted" to job file.
String queue = conf.get(MRJobConfig.QUEUE_NAME,
    JobConf.DEFAULT_QUEUE_NAME); //默认作业调度队列名为"default"
AccessControlList acl = submitClient.getQueueAdmins(queue);
conf.set(toFullPropertyName(queue,
    QueueACL.ADMINISTER_JOBS.getAclName()), acl.getAclString());

// removing jobtoken referrals before copying the jobconf to HDFS
// as the tasks don't need this setting, actually they may break
// because of it if present as the referral will point to a different job.
TokenCache.cleanUpTokenReferral(conf);

if (conf.getBoolean(MRJobConfig.JOB_TOKEN_TRACKING_IDS_ENABLED,
    MRJobConfig.DEFAULT_JOB_TOKEN_TRACKING_IDS_ENABLED)) {
    // Add HDFS tracking ids,如果启用了跟踪机制的话
    ArrayList<String> trackingIds = new ArrayList<String>();
    for (Token<?extends TokenIdentifier> t : job.getCredentials().getAllTokens()) {
        trackingIds.add(t.decodeIdentifier().getTrackingId());
    }
    conf.setStrings(MRJobConfig.JOB_TOKEN_TRACKING_IDS,

```

```

        trackingIds.toArray(new String[trackingIds.size()]));
    } //end if
    ;

    // Write job file to submit dir
    writeConf(conf, submitJobFile); //将 conf 的内容写入一个.xml 文件

    // Now, actually submit the job (using the submit name) /* 万事俱备,只欠提交了 */
    printTokens(jobId, job.getCredentials());
    status = submitClient.submitJob(jobId, submitJobDir.toString(), job.getCredentials());
    //提交作业,通过 YarnRunner.submitJob()或 LocalJobRunner.submitJob()
    if (status != null) {
        return status; //提交成功
    } else {
        throw new IOException("Could not launch job"); //提交失败
    }
} finally {
    if (status == null) { //如果失败就需要善后
        LOG.info("Cleaning up the staging area " + submitJobDir);
        if (jtFs != null && submitJobDir != null) jtFs.delete(submitJobDir, true); //删除目录
    }
} //end try - finally
}

```

代码中已经加了一些注释供读者自己阅读,我们在这里集中关注几个事,那就是 copyAndConfigureFiles()、writeSplits()、writeConf()及最后的 submitClient.submitJob()。

先看 copyAndConfigureFiles()。以前,我们只是笼统地讲提交作业时要将有关的各种资源随同作业单一起提交,但是现在需要具体化了。需要随同作业单一起提交的资源和信息有两类:一类是需要交到资源管理器 RM 手里,供 RM 在立项和调度时使用的;另一类则并非供 RM 直接使用,而是供具体进行计算的节点使用的。前者包括本节点即作业提交者的 IP 地址、节点名、用户名、作业 ID 号,以及有关 MapReduce 计算输入数据文件的信息,还有为提交作业而提供的“证章(Token)”等。这些信息将被打包提交给 RM,这就是狭义的作业提交,是流程的主体。后者则有作业执行所需的 jar 可执行文件、外来对象库等。如果计算的输入文件在本地,则后者还应包括输入文件。这些资源并不需要提交给 RM,因为 RM 本身并不需要用到这些资源,但是必须要把这些资源复制或转移到全局性的 HDFS 文件系统中,让具体承担计算任务的节点能够取用。

为了上传相关的资源和信息,需要在 HDFS 文件系统中为本作业创建一个目录。HDFS 文件系统中有一个目录是专门用于作业提交的,称为“舞台目录(staging directory)”。所以这里要通过 JobSubmissionFiles.getStagingDir()从集群获取这个目录的路径。然后就以本作业的 ID,即 jobId 为目录名在这个舞台目录中创建一个临时的子目录,这就是代码中的 submitJobDir。以后凡是与本作业有关的资源和信息,就都上传到这个子目录中。

首先是通过 `copyAndConfigureFiles()` 上传作业的可执行映像:

```
[Job.submit()>JobSubmitter.submitJobInternal()>copyAndConfigureFiles()]
```

```
copyAndConfigureFiles(Job job, Path jobSubmitDir)
```

```
> conf = job.getConfiguration()
> replication = (short)conf.getInt(Job.SUBMIT_REPLICATION, 10)
> copyAndConfigureFiles(Job job, Path jobSubmitDir, short replication)//多了一个参数
>> String files = conf.get("tmpfiles");//来自命令行中的 -files 选项,计算中需要用到的文件
>> String libjars = conf.get("tmpjars");//来自命令行中的 -libjars 选项,计算中需要用到的 jar
>> String archives = conf.get("tmparchives") //来自命令行中的 -archives 选项
>> String jobJar = job.getJar();//本作业的 java 程序经编译生成的 jar 文件,这是一定有的
>> FileSystem.mkdirs(jtFs, submitJobDir, mapredSysPerms) //在 HDFS 文件系统中创建目录
>> Path filesDir = JobSubmissionFiles.getJobDistCacheFiles(submitJobDir)
>> Path archivesDir = JobSubmissionFiles.getJobDistCacheArchives(submitJobDir)
>> Path libjarsDir = JobSubmissionFiles.getJobDistCacheLibjars(submitJobDir)
>> if (files != null) { //如果命令行中有 -files 选项
>>+ FileSystem.mkdirs(jtFs, filesDir, mapredSysPerms) //在 HDFS 文件系统中创建目录
>>+ String[] fileArr = files.split(",")
>>+ for (String tmpFile: fileArr) {
>>++ tmpURI = new URI(tmpFile)
>>++ tmp = new Path(tmpURI)
>>++ newPath = copyRemoteFiles(filesDir, tmp, conf, replication) //把文件复制到 HDFS 中
>>++> FileSystem remoteFs = originalPath.getFileSystem(conf) //这是本地的一个目录
>>++> if (compareFs(remoteFs, jtFs)) return originalPath //如果源和目的相同就无须复制
>>++> newPath = new Path(parentDir, originalPath.getName()) //准备用在 HDFS 中的目录名
>>++> FileUtil.copy(remoteFs, originalPath, jtFs, newPath, false, conf)
//复制文件,remoteFs 和 originalPath 为源, jtFs 和 newPath 为目标,不删除源文件
>>++> jtFs.setReplication(newPath, replication) //设置备份数量
>>++> return newPath // copyRemoteFiles() 结束
>>++ pathURI = getPathURI(newPath, tmpURI.getFragment())
>>++ DistributedCache.addCacheFile(pathURI, conf)
>>+ }
>> } //end if (files != null)
>> if (libjars != null) { //如果命令行中有 -libjars 选项
>>+ FileSystem.mkdirs(jtFs, libjarsDir, mapredSysPerms) //在 HDFS 文件系统中创建目录
>>+ ...
>>+ newPath = copyRemoteFiles(libjarsDir, tmp, conf, replication) //复制文件至 HDFS 中
>>+ ...
>> }
>> if (archives != null) { //如果命令行中有 -archives 选项
```

```

>>>+ FileSystem.mkdirs(jtFs, archivesDir, mapredSysPerms)
>>>+ ...
>>>+ newPath = copyRemoteFiles(archivesDir, tmp, conf, replication) //复制文件至 HDFS 中
>>>+ ...
>>> }
> if (jobJar != null) { // copy jar to JobTracker's fs,这是无条件的,每个作业都有 jar 文件
>>>+ if (".".equals(job.getJobName())) job.setJobName(new Path(jobJar).getName())
>>>+ jobJarPath = new Path(jobJar)
>>>+ jobJarURI = jobJarPath.toUri()
>>>+ if (jobJarURI.getScheme() == null || jobJarURI.getScheme().equals("file")) {
>>>+ copyJar(jobJarPath, JobSubmissionFiles.getJobJar(submitJobDir), replication)
>>>+> jtFs.copyFromLocalFile(originalJarPath, submitJarFile)
>>>+> jtFs.setReplication(submitJarFile, replication)
>>>+> jtFs.setPermission(submitJarFile,
new FsPermission(JobSubmissionFiles.JOB_FILE_PERMISSION));
>>>+ job.setJar(JobSubmissionFiles.getJobJar(submitJobDir).toString())
>>>+ }
>>> }

```

这个函数要做的事情,首先是 copy,就是把作业提交者所在本地(宿主)文件系统中的一些文件复制到 HDFS 中,因为 HDFS 是全局的、谁都能访问到的,而且有多个复份。那么要复制一些什么文件呢?首先就是作业本身的可执行程序,就是编译产生的 jar 文件。除此之外,在启动这个作业的命令中也可能以可选项的方式提供一些文件,那也要复制到 HDFS 中。HDFS 文件不存在“在哪个节点上”的问题,同一个文件中不同的“块”就可以在不同的节点上。注意,submitJobInternal()所直接调用的是两个参数的 copyAndConfigureFiles(),这个函数补上一个参数 replication 即复份个数,再调用三个参数的 copyAndConfigureFiles(),以完成操作。复制到 HDFS 的文件会被保存多份(一般是三份),但是可以因具体文件而不同。那么这些文件要保存几份呢?可以在配置块中设定,这里默认的是 10 份。10 个复份,会被存储在 10 个不同的节点上,看起来似乎很多,但是考虑到运行时 Mapper 的数量可能会有数百上千,那也就不稀奇了。不过,要是 Mapper 的数量只有三五个,那这个开销所占的比例就太高了。由此可见,把很小的题目放在 Hadoop 上计算是不划算的。

代码摘要中加了注释,这里就不多作讲解了。注意,注释原文中提到的 JobTracker 是老版本中才有的,现在已为 ResourceManager 和 ApplicationMaster 所替代。

回到前面 submitJobInternal()的摘要中,下一步是生成并上传关于输入数据分片的信息,即 split 文件,这是由 writeSplits()完成的:

```
[Job.submit()>JobSubmitter.submitJobInternal()> writeSplits()]
```

```
JobSubmitter.writeSplits(JobContext job, Path jobSubmitDir)
```

```

> jConf = (JobConf) job.getConfiguration()
> if (jConf.getUseNewMapper()) {

```



```

>+ int maps = writeNewSplits(job, jobSubmitDir)
                                //按新 API 的要求写 Split 文件,返回 Split 的数量
>+> InputFormat<?,?> input = ReflectionUtils.newInstance(job.getInputFormatClass(), conf)
>+> List<InputSplit> splits = input.getSplits(job) == FileInputFormat.getSplits(job)
                                //为输入(数据)文件生成一个 InputSplit 的 List
>+>> minSize = Math.max(getFormatMinSplitSize(), getMinSplitSize(job)) //最小 Split 尺寸
>+>> maxSize = getMaxSplitSize(job) //最大 Split 尺寸
>+>> List<InputSplit> splits = new ArrayList<InputSplit>() //创建一个空白的 Split List
>+>> List<FileStatus> files = listStatus(job) //获取每个输入文件的 FileStatus
>+>> for (FileStatus file: files) //对于每个输入文件(可以不止一个)
>+>>+ path = file.getPath() //获取其路径
>+>>+ length = file.getLen() //获取其文件长度
>+>>+ if (length != 0) { //如果文件长度不为 0
>+>>++ if (file instanceof LocatedFileStatus) { //如果是个含有数据块位置信息的文件
>+>>+++ blkLocations = ((LocatedFileStatus) file).getBlockLocations()
>+>>+++ } else { //一般的文件
>+>>+++ FileSystem fs = path.getFileSystem(job.getConfiguration())
>+>>+++ blkLocations = fs.getFileBlockLocations(file, 0, length)
>+>>+++ }
>+>>++ if (isSplittable(job, path)) {
>+>>+++ blockSize = file.getBlockSize()
>+>>+++ splitSize = computeSplitSize(blockSize, minSize, maxSize)
                                //Split 的大小不一定是数据块的大小,但通常都是
>+>>+++ bytesRemaining = length
>+>>+++ while (((double) bytesRemaining)/splitSize > SPLIT_SLOP) {
>+>>++++ blkIndex = getBlockIndex(blkLocations, length - bytesRemaining)
>+>>++++ splits.add(makeSplit(path, length - bytesRemaining, splitSize,
                                blkLocations[blkIndex].getHosts(),
                                blkLocations[blkIndex].getCachedHosts()))
>+>>++++ bytesRemaining -= splitSize
>+>>+++ }
>+>>+++ if (bytesRemaining != 0) { //剩下的尾巴(剩余部分)作为一个分片
>+>>++++ blkIndex = getBlockIndex(blkLocations, length - bytesRemaining)
                                //分片起点所在数据块的 index
>+>>++++ splits.add(makeSplit(path, length - bytesRemaining, bytesRemaining,
                                blkLocations[blkIndex].getHosts(),
                                blkLocations[blkIndex].getCachedHosts()))
>+>>+++ }
>+>>+ } else { //not splittable
>+>>+++ splits.add(makeSplit(path, 0, length, blkLocations[0].getHosts(),

```

```

        blkLocations[0].getCachedHosts()))
    >+>> ++ }
    >+>> + } else { //文件长度为 0
    >+>> ++ splits.add(makeSplit(path, 0, length, new String[0]))
    >+>> + }
    >+>> } //end for each file
    >+>> job.getConfiguration().setLong(NUM_INPUT_FILES, files.size())
    >+>> return splits
    >+> array = (T[]) splits.toArray(new InputSplit[splits.size()])
    >+> Arrays.sort(array, new SplitComparator())
    >+> JobSplitWriter.createSplitFiles(jobSubmitDir, conf,
        jobSubmitDir.getFileSystem(conf), array) //创建 Split 文件
    >+> return array.length
    > } else {
    >+ maps = writeOldSplits(jConf, jobSubmitDir)
    > }
    > return maps

```

如前所述,把输入数据文件分成多少个切片(Split),将来就会安排多少个 Mapper 分头进行计算。那么这个输入数据文件是根据什么条件来分片的呢? 根据大小。配置文件 mapred-default.xml 中应该有两项配置: “mapreduce.input.fileinputformat.split.minsize”, 即 SPLIT_MINSIZE; “mapreduce.input.fileinputformat.split.maxsize”, 即 SPLIT_MAXSIZE。如果没有就用默认值。但是切片的大小还有别的限制,那就是 Hadoop 文件系统中“块(block)”的大小,一个块的大小一般是 128MB。

这里摘要中的 splits 就是对分片信息的描述。这是个 InputSplit 对象的 List,不过 InputSplit 是抽象类,因为这里要考虑输入数据可能有多种不同的来源,比方说来自数据文件,也可以来自数据库查询的输出,也可能实时来自网络。来源不同,分片的方法自然也就不同。就数据文件而言,扩充了 InputSplit 的具体类为 FileSplit:

```

class FileSplit extends InputSplit implements Writable {
    ] Path file //输入文件路径
    ] long start //分片在文件中的位置(起点)
    ] long length //分片长度
    ] String[] hosts //这个分片所在数据块的多个复份所在节点
    ] SplitLocationInfo[] hostInfos //每个数据块复份所在节点,以及是否缓存
    ]] boolean inMemory //是否缓存在内存中
    ]] String location //所在节点

```

有了这么一个 List 之后,就将其转化成一个数组,并加以排序,然后就将其串行化后写入文件,以供作业提交之用。注意,我们在这里所谓的一个 Split、一个 FileSplit,只是对一个 Split 的描述,而不是这个 Split 本身。至于 FileSplit 所描述的某个 HDFS 文件的那部分内容,则已经重复存储在某几个节点上。

```
[Job.submit()>JobSubmitter.submitJobInternal() > writeSplits() >writeNewSplits()
>JobSplitWriter.createSplitFiles()]
```

```
createSplitFiles(jobSubmitDir, conf, fs, array)
```

```
> path1 = JobSubmissionFiles.getJobSplitFile(jobSubmitDir) //路径名为“~/job.split”
```

```
> FSDataOutputStream out = createFile(fs, path1, conf)
```

```
//创建输入片(Split)文件,并为其创建一个输出流
```

```
> SplitMetaInfo[] info = writeNewSplits(conf, splits, out) //将分片信息写入 Split 文件:
```

```
>> info = new SplitMetaInfo[array.length]
```

```
>> if (array.length!= 0) {
```

```
>>>+ factory = new SerializationFactory(conf)
```

```
>>>+ maxBlockLocations = conf.getInt(MRConfig.MAX_BLOCK_LOCATIONS_KEY,
```

```
MRConfig.MAX_BLOCK_LOCATIONS_DEFAULT)
```

```
>>>+ offset = out.getPos()
```

```
>>>+ for(T split: array) { //对于数组中的每一个 Split:
```

```
>>>+ prevCount = out.getPos()
```

```
>>>+ Text.writeString(out, split.getClass().getName())
```

```
>>>+ Serializer<T> serializer = factory.getSerializer((Class<T>) split.getClass())
```

```
>>>+ serializer.open(out) //串行化后的内容会写入输出流 out
```

```
>>>+ serializer.serialize(split)
```

```
>>>+ currCount = out.getPos()
```

```
>>>+ String[] locations = split.getLocations()
```

```
>>>+ if (locations.length > maxBlockLocations) {
```

```
>>>+ locations = Arrays.copyOf(locations, maxBlockLocations)
```

```
>>>+ }
```

```
>>>+ info[i++] = new JobSplit.SplitMetaInfo(locations, offset, split.getLength())
```

```
>>>+ offset += currCount - prevCount
```

```
>>>+ } //end for
```

```
>> }
```

```
>> return info
```

```
> out.close()
```

```
> path2 = JobSubmissionFiles.getJobSplitMetaFile(jobSubmitDir) //“~/job.splitmetainfo”
```

```
> writeJobSplitMetaInfo(fs, path2,
```

```
new FsPermission(JobSubmissionFiles.JOB_FILE_PERMISSION), splitVersion,info)
```

```
>> FSDataOutputStream out = FileSystem.create(fs, filename, p)
```

```
>> out.write(JobSplit.META_SPLIT_FILE_HEADER)
```

```
>> WritableUtils.writeVInt(out, splitMetaInfoVersion)
```

```
>> WritableUtils.writeVInt(out, allSplitMetaInfo.length) // allSplitMetaInfo 就是上面的 info[]
```

```
>> for (JobSplit.SplitMetaInfo splitMetaInfo : allSplitMetaInfo) {
```

```
>>>+ splitMetaInfo.write(out)
```

```
>> }
>> out.close()
```

这里生成了两个文件,都在为具体作业创建的目录下,第一个是“job.split”,第二个是“job.splitmetainfo”。前者的内容是对于每个 Split 描述的记录,相当于一个数组,但是每个元素的长度可能有所不同(因为 String)。注意,在大型的计算中可能会有数百、上千个 Split。后者为元数据文件,相当于为前者所做的索引,其信息来自写第一个文件时的积累。这里的 allSplitMetaInfo 就是上面的 SplitMetaInfo 数组 info[], SplitMetaInfo 类定义为:

```
class SplitMetaInfo implements Writable {
    ] long startOffset          //这是在 Split 文件即“job.split”中的位移
    ] long inputDataLength
    ] String[] locations
```

再回到前面 submitJobInternal()的摘要中,最后,还要通过 writeConf()把配置块 conf 的内容加以“串行化”并写入一个.xml 文件,当然这也在 RM 节点上本作业的子目录中:

```
writeConf(conf, submitJobFile)
> FSDDataOutputStream out = FileSystem.create(jtFs, jobFile,
                                                newFsPermission(JobSubmissionFiles.JOB_FILE_PERMISSION))
> conf.writeXml(out)
> out.close()
```

提交作业时还有个提交给 RM 的哪一个队列的问题,这跟作业的调度运行有关。RM 有三种不同的调度策略,即“先进先出”、“基于功能与容量”、“基于公平”,从而就有三个不同的队列。那么怎么确定呢?目前是通过.xml 配置文件静态设置的,需要以 QUEUE_NAME,即“mapreduce.job.queue.name”为键值去配置块中查询(实际来自文件 mapred-default.xml)。此外,那些队列是有访问控制名单(ACL)的,并非任何用户都可以将作业提交给任何一个队列,所以还要提供与 ACL 相关的身份信息,即 Credentials。

至此,已是万事俱备,只欠提交了,而提交是 submitClient.submitJob()的事。

这个 submitClient 是 JobSubmitter 内部的一个成分,是实现了 ClientProtocol 界面的 YARNRunner 或 LocalJobRunner 对象。至于具体是哪一种,则要看配置文件中对于“mapreduce.framework.name”这一项的设定,或者在启动作业运行时命令行中的相应参数设定。对于 Hadoop 集群,这应该是 YARNRunner(如果是在单机上,则是 LocalJobRunner。至于 2.0 版之前的 Hadoop 那是另一回事了)。这是在前面 Cluster.Initialize()中确定并创建的。

于是,这个作业的 JobId、相关文件所在的目录路径 submitJobDir 和身份信息 Credentials,就将由 YARNRunner 提交给 RM 节点。

5.6 YARNRunner 和 ResourceMgrDelegate

YARN,是“Yet Another Resource Negotiator”的缩写,意为“又一个资源协调者”,知道 C 编译和 YACC 的人马上就会联想到“Yet Another C Compiler”。之所以是“又一个”,是因为 Hadoop 原先在 0.x 和 1.x 版中就有个类似于“资源协调者”这样的角色,只是不叫这个名称,

所用的机制也不一样,然后又开发出了一种结构更好、概念更清晰、功能也更强的(集群)资源协调/管理机制,所以才叫 YARN。不过也有人主张 YARN 是“YARN Application Resource Negotiator”的缩写,这又使人联想到 GNU 是“GNU is Not Unix”的缩写,这是一样的套路。

而 YarnRunner,顾名思义就是 YARN 的 Runner。我们先看一下这个类的摘要:

```
class YARNRunner implements ClientProtocol{
    [ ResourceMgrDelegate resMgrDelegate //这是 RM 派驻在“地方”上的特派员
    [ ClientCache clientCache
    [ FileContext defaultFileContext
    [ YARNRunner(Configuration conf) //构造函数
        > this(conf, new ResourceMgrDelegate(new YarnConfiguration(conf)))
            //需要创建特派员,然后调用下一个构造函数
    [ YARNRunner(Configuration conf, ResourceMgrDelegate resMgrDelegate) //构造函数
        > this(conf, resMgrDelegate, new ClientCache(conf, resMgrDelegate))
            //需要创建 ClientCache,然后调用下一个构造函数
    [ YARNRunner(Configuration conf, ResourceMgrDelegate resMgrDelegate,
        ClientCache clientCache) //这是最终的构造函数
        > this.resMgrDelegate = resMgrDelegate
        > this.clientCache = clientCache
        > this.defaultFileContext = FileContext.getFileContext(this.conf)
    [ submitJob(JobID jobId, String jobSubmitDir, Credentials ts)
    [ createApplicationResource(FileContext fs, Path p, LocalResourceType type)
    [ createApplicationSubmissionContext(Configuration jobConf, String jobSubmitDir, ...)
    [ killJob(JobID arg0)
    [ killTask(TaskAttemptID arg0, boolean arg1)
```

YARNRunner 内部有个起着重要作用的 ResourceMgrDelegate 类对象 resMgrDelegate,这是在创建 YARNRunner 时在其构造方法中创建的,或者是作为构造参数之一传下来的。与此相似,还有个 ClientCache 类对象 clientCache,那也是在创建 YARNRunner 的时候创建的,但是重要性就没有那么大了。

我们先来看 YARNRunner.submitJob()的代码,这已是“临门一脚”了。

```
[Job.submit() > JobSubmitter.submitJobInternal() > YARNRunner.submitJob()]

public JobStatus submitJob(JobID jobId, String jobSubmitDir, Credentials ts) throws ...{
    addHistoryToken(ts); //用于为历史记录服务,与“作业历史(JobHistory)”有关
    // Construct necessary information to start the MR AM(AM 是 ApplicationMaster)
    ApplicationSubmissionContext appContext =
        createApplicationSubmissionContext(conf, jobSubmitDir, ts);
    //创建一个 ApplicationSubmissionContext,并将 conf 中的相关信息转移过去

    // Submit to ResourceManager /* 将作业提交给资源管理者(ResourceManager) */
```

```

try {
    ApplicationId applicationId = resMgrDelegate.submitApplication(appContext);
    //由 RM 派驻的特派员转交 ApplicationSubmissionContext
    ApplicationReport appMaster = resMgrDelegate.getApplicationReport(applicationId);
    String diagnostics =
        (appMaster == null?"application report is null" : appMaster.getDiagnostics());
    if (appMaster == null
        || appMaster.getYarnApplicationState() == YarnApplicationState.FAILED
        || appMaster.getYarnApplicationState() == YarnApplicationState.KILLED) {
        throw new IOException("Failed to run job : " + diagnostics);
    }

    return clientCache.getClient(jobId).getJobStatus(jobId);
} catch (YarnException e) {
    throw new IOException(e);
} // end try - catch
}

```

所谓作业提交,是要把作业交到 ResourceManager 即 RM 的手里。RM 是整个集群的资源管理者。更确切地说,这是一个 ResourceManager 类的对象,这个对象在主节点上。整个集群中只有一个实际有效的 ResourceManager 对象,其所在的节点可以通过.xml 配置文件加以指定,也可以通过在某一机器节点上启动 ResourceManage 而使其成为事实上的主节点。除这个节点之外,其余每个节点上都有个 NodeManager,更确切地说是一个 NodeManager 类对象,这个对象起着类似于地方政府的作用。不过,正如我们在前面的代码中所见,向“中央”提交作业并不需要通过“地方政府”;但是,当“中央”为了完成作业而要向“地方”指派任务的时候,就要跟 NodeManager 打交道了。所以,NodeManager 本质上也是一种 Server。至于 ResourceManager,虽然统管着整个集群的资源 and 所有作业的调度运行,对于 NodeManager 而言却是 Client;但是,相对于用户,相对于具体的应用,例如 WordCount 和 QuasiMonteCarlo 这些应用,却又是 Server。所以 Server 和 Client 的概念有层次的不同,要看是在什么层次上说。

那么,提交物又是什么呢?我们在前面看到,与作业有关的资源已经上传到全局的 HDFS 文件系统中,那里已经有了一个专用于本作业的临时目录,所缺的就是一个类似于“申请报告”、“项目说明书”这样的东西了。ASC,即 ApplicationSubmissionContext 就起着这样的作用,所谓的 Context 本来就有“来龙去脉”、“上下文”的意思。所以这里通过 createApplicationSubmissionContext()创建了一份 ApplicationSubmissionContext,并把配置块 conf 中当前的相关信息、已上传资料所在的目录路径以及有关身份和访问权限的信息都复制转移过去。所谓 Application,就其本质而言,其实就是 Job,只是具体的数据结构有所不同,信息的范围又有所扩大而已。而所谓 Context,又接近于 Configuration,但也有所扩大,增加的那些信息基本上也是来自配置块。例如,我们得告诉 ResourceManager 需要多大的内存,这就要通过字符串 MR_AM_VMEM_MB,即“yarn.app.mapreduce.am.resource.mb”在配置块中查询。如果配置文件中未作规定,就采用默认值 1536MB。此外,还要告诉 ResourceManager

需要几个 CPU 核来执行我们的作业,这就要通过字符串 MR_AM_CPU_VCORES,即“yarn.app.mapreduce.am.resource.cpu-vcores”在配置块中查询,如果配置文件中未作规定,就采用默认值 1。特别地,这个函数还提供了有关 Application Master 即“项目组长”该用哪一个 Shell (例如 bash) 以及有关某些环境变量的信息。再如作业的名称,如此等等,不一而足。createApplicationSubmissionContext() 这个函数的代码烦琐但并不复杂,就留给读者自己阅读理解了。重要的是,ApplicationSubmissionContext,即 appContext,才是按 ResourceManager 所要求模板填写的“正式作业单”。按源文件 ApplicationSubmissionContext.java 中的注释,这个类的对象“代表着 ResourceManager 为发起该应用的 ApplicationMaster 所需的全部信息”。

不过,ApplicationSubmissionContext 其实是个抽象类,具体加以落实的类是 ApplicationSubmissionContextPBImpl,所以实际创建和提交的作业单是后一类的对象。这里的 PB 是 Protocol Buffer 的缩写,本书前面已有介绍。显然,ApplicationSubmissionContextPBImpl 意为 ApplicationClientProtocol 采用 PB 技术的实现。这个类的数据部分摘要如下:

```
class ApplicationSubmissionContextPBImpl extends ApplicationSubmissionContext {}
] ApplicationSubmissionContextProto proto    //这是通信协议部分
] ApplicationId applicationId
] Priority priority
] ContainerLaunchContext amContainer
] Resource resource
] Set<String> applicationTags
] ResourceRequest amResourceRequest
] LogAggregationContext logAggregationContext
] ReservationId reservationId              //用于资源预订
```

这里面如 applicationId、priority、amResourceRequest 等成分的作用不言自明。其实最重要的成分倒是 amContainer,这是一个 CLC,即 ContainerLaunchContext 对象,是 RM 为这个应用指派建立 ApplicationMaster 时所给定的 Context。这个 Context 决定着本应用中各个任务的执行。显然,这些信息的源头都在提交作业的这一方。同样,这也是个抽象类,具体予以落实的类是 ContainerLaunchContextPBImpl:

```
class ContainerLaunchContextPBImpl extends ContainerLaunchContext {}
] ContainerLaunchContextProto proto    //这是通信协议部分
] Map<String, LocalResource> localResources
] ByteBuffer tokens
] Map<String, ByteBuffer> serviceData
] Map<String, String> environment
] List<String> commands    //这是在指派用于本作业 AM 的节点上启动 AM 的命令行
] Map<ApplicationAccessType, String> applicationACLs
```

看似貌不惊人,其实所含的信息量是很大的。特别值得一提的是其中的 commands,这是供 RM 在某个节点上发起一个 ApplicationMaster 进程时所用的命令行。我们在前面看到 YARNRunner.submitJob() 调用了一个函数 createApplicationSubmissionContext(),这个函

数的代码中就有这么一个片段：

```
// Setup the command to run the AM
List<String> vargs = new ArrayList<String>(8);
vargs.add(MRApps.crossPlatformifyMREnv(jobConf, Environment.JAVA_HOME)
                                                + "/bin/java");
... //这中间是许多命令行选项和参数
vargs.add(MRJobConfig.APPLICATION_MASTER_CLASS);
//这是“org.apache.hadoop.mapreduce.v2.app.MRAppMaster”
vargs.add("1>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR +
          Path.SEPARATOR + ApplicationConstants.STDOUT); //这是“<LOG_DIR>/stdout”
vargs.add("2>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR +
          Path.SEPARATOR + ApplicationConstants.STDERR); //这是“<LOG_DIR>/stderr”
```

这段程序在运行时生成的是下面这样的一个 Shell 命令行：

```
JAVA_HOME/bin/java ...org.apache.hadoop.mapreduce.v2.app.MRAppMaster \
1> <LOG_DIR>/stdout 2> <LOG_DIR>/stderr
```

这里的<LOG_DIR>为日志文件所在目录，应以实际的目录路径代入。

RM 受理了所提交的作业以后，会把这个 ContainerLaunchContext 转发到某个 NM 节点上，在那里执行这个 shell 命令行，另起一个 Java 虚拟机，让它执行 MRAppMaster.class。

由此可见，这个 ApplicationSubmissionContext 对象 appContext，真的是“代表着 ResourceManager 为发起该应用的 ApplicationMaster 所需的全部信息”。

而 appContext 的提交，则是通过 ResourceMgrDelegate.submitApplication() 完成的。ResourceMgrDelegate，顾名思义是 ResourceManager 的 Delegate，即“派驻代表”、“特派员”。从概念上说，既然是“代表”，YARNRunner 通过 ResourceMgrDelegate.submitApplication() 把 appContext 交到了它的手里，就算是提交给了 ResourceManager，以后就是“代表”怎么把 appContext 送达 RM 的事了。我们先看一下 ResourceMgrDelegate 类的摘要，这个类是对抽象类 YarnClient 的继承和扩展：

```
class ResourceMgrDelegate extends YarnClient{}
] YarnConfiguration conf
] ApplicationSubmissionContext application
] YarnClient client //实际上是 YarnClientImpl 类的对象，那也是对 YarnClient 的继承和扩展
] ResourceMgrDelegate(YarnConfiguration conf) //这是 ResourceMgrDelegate 的构造方法
> super(ResourceMgrDelegate.class.getName())
> this.conf = conf
> this.client = YarnClient.createYarnClient() //创建 YarnClient 对象 client
>> client = new YarnClientImpl() // YarnClient.createYarnClient()创建的是 YarnClientImpl
> init(conf) //这是由 AbstractService 类提供的，YarnClient 是对 AbstractService 的扩展
>> serviceInit(config)
>> notifyListeners()
```

```

> start() //这也是由 AbstractService 类提供的
>> stateModel.enterState(STATE.STARTED)
>> startTime = System.currentTimeMillis()
>> serviceStart()
>> notifyListeners()
] serviceInit(Configuration conf)
> client.init(conf) == YarnClientImpl.init(conf)
>> AbstractService.init()
>>> serviceInit() == YarnClientImpl.serviceInit()
>>>> ...
> super.serviceInit(conf)
] serviceStart()
> client.start() == YarnClientImpl.start()
>> AbstractService.start()
>>> serviceStart() == YarnClientImpl.serviceStart()
>>>> rmClient = ClientRMProxy.createRMProxy(..., ApplicationClientProtocol.class)
>>>> historyClient.start()
>>>> timelineClient.start()
>>>> super.serviceStart()
>>> notifyListeners()
> super.serviceStart()
] submitApplication(ApplicationSubmissionContext appContext)
> client.submitApplication(appContext)

```

我们在前面看到, ResourceMgrDelegate 对象是在 YARNRunner 的构造函数中创建的。而 YARNRunner, 则是在前面的 Cluster.Initialize() 中创建的。再往上追溯, 则 Cluster 类对象是在首次调用 connect() 时创建的。所以, 任何一个节点, 只要曾经调用过 connect(), 即曾经与“集群”有过连接, 节点上就会有 Cluster 类对象, 从而就会有 YARNRunner 对象, 也就会有 ResourceMgrDelegate 对象, 而且如下所述就会有 YarnClientImpl 对象。

从 ResourceMgrDelegate 类的摘要可以看出, 它的 submitApplication() 方法其实十分简单, 只是转而调用 client.submitApplication() 而已。而 ResourceMgrDelegate 类中的 client, 则同样可以从 ResourceMgrDelegate 类的摘要中看出, 那是在其构造函数 ResourceMgrDelegate() 中通过 YarnClient.createYarnClient() 创建的, 类型为 YarnClientImpl。

像 ResourceMgrDelegate 一样, YarnClientImpl 也是对 YarnClient 的继承和扩充, 当然是不同的扩充。YarnClientImpl, 顾名思义就是对 YarnClient 的具体实现 (implimentation)。而 YARNRunner.submitJob() 对 ResourceMgrDelegate.submitApplication() 的调用, 则实际上落实为对于 YarnClientImpl.submitApplication() 的调用。

```

[Job.submit() > JobSubmitter.submitJobInternal() > YARNRunner.submitJob()
> ResourceMgrDelegate.submitApplication() > YarnClientImpl.submitApplication()]

```

```

public ApplicationId submitApplication(ApplicationSubmissionContext appContext)
                                throws YarnException, IOException {
    ApplicationId applicationId = appContext.getApplicationId();
    if (applicationId == null) {
        throw new ApplicationIdNotProvidedException(
            "ApplicationId is not provided in ApplicationSubmissionContext");
    }
    // 创建一个 SubmitApplicationRequestPBImpl 类的记录块
    SubmitApplicationRequest request =
        Records.newRecord(SubmitApplicationRequest.class);
    request.setApplicationSubmissionContext(appContext); // 设置好记录块中的 Context

    // Automatically add the timeline DT into the CLC
    // Only when the security and the timeline service are both enabled
    if (isSecurityEnabled() && timelineServiceEnabled) {
        addTimelineDelegationToken(appContext.getAMContainerSpec());
    }

    // TODO: YARN-1763: Handle RM failovers during the submitApplication call.
    rmClient.submitApplication(request); // 实际的跨节点提交

    int pollCount = 0;
    long startTime = System.currentTimeMillis();

    while (true) {
        try {
            YarnApplicationState state =
                getApplicationReport(applicationId).getYarnApplicationState();
            // 获取来自 RM 节点的应用状态报告, 从中获取本应用的当前状态
            if (!state.equals(YarnApplicationState.NEW) &&
                !state.equals(YarnApplicationState.NEW_SAVING)) {
                LOG.info("Submitted application " + applicationId);
                break; // 作业已进入运行阶段, 结束 while 循环
            }
        }

        long elapsedMillis = System.currentTimeMillis() - startTime;
        if (enforceAsyncAPITimeout() && elapsedMillis >= asyncApiPollTimeoutMillis) {
            throw new YarnException("Timed out while waiting for application " +
                applicationId + " to be submitted successfully");
        }
    }
}

```

```

    // Notify the client through the log every 10 poll, in case the client
    // is blocked here too long.
    if ( ++ pollCount % 10 == 0 ) {
        LOG.info("Application submission is not finished, " +
            "submitted application " + applicationId + " is still in " + state);
    }
    try {
        Thread.sleep(submitPollIntervalMillis); //睡眠一段时间
    } catch (InterruptedException ie) {
        LOG.error("Interrupted while waiting for application " + applicationId
            + " to be successfully submitted.");
    }
} catch (ApplicationNotFoundException ex) {
    // FailOver or RM restart happens before RMStateStore saves ApplicationState
    LOG.info("Re-submit application " + applicationId + "with the " +
        "same ApplicationSubmissionContext");
    rmClient.submitApplication(request); //失败后的再次提交
} //end try-catch
} //end while,进入下一轮循环
return applicationId;
}

```

实际跨节点提交之前先把 appContext 转化成是一个 SubmitApplicationRequest 记录块,然后通过 rmClient.submitApplication() 提交请求。由于是跨节点的通信,在发出请求之后就进入一个 while 循环,不时地通过 getApplicationReport() 询问对方并接收对岸发回的报告。这两种可能的结果:一种是收不到报告,发生了 ApplicationNotFoundException 异常,那就再发一遍,然后再 getApplicationReport(); 另一种是收到了报告,那就要从报告中查看所提交的应用处于什么状态。如果已经不在 NEW 或 NEW_SAVING 状态,就说明已经开始运行,那就可以跳出循环并返回了;否则就睡眠一会儿,等一下再来 getApplicationReport()。

对我们来说,这个流程还没有走到头,还要进一步追问 rmClient.submitApplication() 是怎么实现的。

这个 rmClient 是内嵌在 YarnClientImpl 中实现了 ApplicationClientProtocol 界面的对象,具体就是 ApplicationClientProtocolPBClientImpl 类对象。ApplicationClientProtocol 是个界面,同时也是一种通信规程(Protocol),或称“协议”,这个界面就是为这种规程而设计和定义的。而 ApplicationClientProtocolPBClientImpl 类则采用 Protocol Buffer 技术实现了这个界面,也就是实现了这个规程。

本书第 4 章介绍过“Protocol Buffer”,即 protobuf,以及用其语言编写的 proto 文件 applicationclient_protocol.proto 的内容,和经过编译工具 protoc 加以编译之后生成的代码。

这个 proto 文件定义了一个名为 ApplicationClientProtocolService 的 Service,即 RPC 服务,编译工具 protoc 就生成实现这个服务的代码,包括服务端和客户端两方的代码。这样,我

们在客户端调用由这种服务所提供的 `submitApplication()` 时,它就向对方发出一个 `SubmitApplicationRequestProto` 报文。而服务端就在那里调用某个类的 `submitApplication()` 方法,并在该方法结束和返回后向请求方发回一个 `SubmitApplicationResponseProto` 报文。这样,表面上是本地的一次 `submitApplication()` 调用,实际上调用的却是远地的同名操作方法。

但是,这里马上就有两个问题:在本地调用的 `submitApplication()` 方法是由什么类提供的?在远地调用的又是什么类的 `submitApplication()` 方法?答案就在于,这个 `service` 定义经 `protoc` 编译就会生成双方这两个类的代码。

首先,.proto 文件中定义的 `Service` 如 `ApplicationClientProtocolService` 其实就是一个界面,而所生成的客户端 `Proxy` 类,即“代理类”,则实现了这个界面,并且又是对 `ApplicationClientProtocol` 的扩充。这个类就提供了 `submitApplication()` 以及所定义的其他方法。向对方发送 `SubmitApplicationRequestProto` 报文就是由这个方法完成的。

另一方面,protoc 也会生成一个运行于对方(服务端)的 `Service` 类,这个类在接收到 `SubmitApplicationRequestProto` 报文时会调用那里某个扩充了 `ApplicationClientProtocol` 的“真实的、实际的”类所提供的 `submitApplication()`。

于是,由 `protoc` 生成的代码对于客户端即操作的请求方就变得“透明”了,我们可以在概念上认为在客户端调用的就是服务端的那个类所提供的 `submitApplication()`,这两个类也正好都是对 `ApplicationClientProtocol` 的扩充。至于过程中的串行化/去串行化等,则都已包含在其中了。

在前面 `YarnClientImpl.submitApplication()` 的代码中,`rmClient` 是 `YarnClientImpl` 内部一个实现了 `ApplicationClientProtocol` 界面的对象。`YarnClientImpl` 在其创建过程中执行其 `serviceStart()` 方法,通过 `ClientRMProxy.createRMProxy()` 创建了这个对象,属于 `ApplicationClientProtocolPBClientImpl` 类。不过 `ApplicationClientProtocolPBClientImpl` 类倒不是直接由 `protoc` 生成的,而只是使用了由 `protoc` 生成的类作为其底层。

```
class YarnClientImpl extends YarnClient {}
] ApplicationClientProtocol rmClient
] AHSCClient historyClient
] YarnClientImpl()
> super() == YarnClient.YarnClient() //执行其父类 YarnClient 的构建方法
>> super() == AbstractService.AbstractService()
] serviceInit(Configuration conf)
] serviceStart()
> rmClient = ClientRMProxy.createRMProxy(getConfig(), ApplicationClientProtocol.class)
> historyClient.start()
> timelineClient.start()
> super.serviceStart()
] createApplication()
] submitApplication(ApplicationSubmissionContext appContext) //见前文所引代码
> rmClient.submitApplication(request)
== ApplicationClientProtocolPBClientImpl.submitApplication()
```


通过这个摘要,我们知道了 rmClient 的来历,也知道了由 YarnClientImpl 提供的操作 submitApplication() 其实是通过 YarnClientImpl.ApplicationClientProtocolPBClientImpl() 实现的,换言之是通过 ApplicationClientProtocolPBClientImpl.submitApplication() 实现的。

我们继续往下追看:

```
[Job.submit() > JobSubmitter.submitJobInternal() > YARNRunner.submitJob() >
ResourceMgrDelegate.submitApplication() > YarnClientImpl.submitApplication() >
ApplicationClientProtocolPBClientImpl.submitApplication()]
```

```
submitApplication(SubmitApplicationRequest request)
```

```
> requestProto = ((SubmitApplicationRequestPBImpl) request).getProto()
//从请求 request 中取出其协议报文(message)部分
> resp = proxy.submitApplication(null, requestProto)
//交由 proxy 将报文发送出去,并等候服务端回应
> return new SubmitApplicationResponsePBImpl(resp)
//将服务端回应包装成 SubmitApplicationResponsePBImpl 对象
```

这里我们也需要知道两个信息:一是这里 proxy 的来历;二是 proxy.submitApplication() 做了些什么。所以我们先重温并看一下 ApplicationClientProtocolPBClientImpl 类的进一步的摘要:

```
class ApplicationClientProtocolPBClientImpl implements ApplicationClientProtocol, ...{}
] ApplicationClientProtocolPB proxy
] ApplicationClientProtocolPBClientImpl(long clientVersion,
                                         InetAddress addr, Configuration conf)
> RPC.setProtocolEngine(conf, ApplicationClientProtocolPB.class,
                                         ProtobufRpcEngine.class)
>> conf.setClass(ENGINE_PROP + ". " + protocol.getName(), engine, RpcEngine.class)
//将配置项“rpc.engine.ApplicationClientProtocolPB”设置成 ProtobufRpcEngine
> proxy = RPC.getProxy(ApplicationClientProtocolPB.class, clientVersion, addr, conf)
>> ProtocolProxy<T> p = RPC.getProtocolProxy(protocol, clientVersion, addr, conf)
>>> if (UserGroupInformation.isSecurityEnabled()) SaslRpcServer.init(conf) //如果需要加密
>>> return getProtocolEngine(protocol, conf).getProxy(protocol, clientVersion, ...)
                                         == ProtobufRpcEngine.getProxy(protocol, clientVersion, ...)
//参数 protocol 为 ApplicationClientProtocolPB
>>>> Invoker invoker = new Invoker(protocol, addr, ticket, conf, factory, rpcTimeout,
                                         connectionRetryPolicy, fallbackToSimpleAuth) //创建 Invoker
>>>> (T) _proxy = Proxy.newProxyInstance(protocol.getClassLoader(),
                                         new Class[] { protocol }, invoker)
>>>> return new ProtocolProxy<T>(protocol, _proxy, false)
//参数 _proxy 被赋给 ProtocolProxy.proxy
>> p.getProxy() == ProtocolProxy<T>.getProxy()
```

```
>>> return proxy//返回值被赋给上面的 ApplicationClientProtocolPBClientImpl.proxy
//这就是由 Proxy.newProxyInstance()创建的那个对象
] submitApplication(SubmitApplicationRequest request) //见上
> requestProto = ((SubmitApplicationRequestPBImpl) request).getProto()
> resp = proxy.submitApplication(null, requestProto)
> return new SubmitApplicationResponsePBImpl(resp)
```

显然,这个 proxy 是在 ApplicationClientProtocolPBClientImpl 类的构造函数中创建的,其类型为 ApplicationClientProtocolPB。这创建的过程还真有点复杂。

先是通过 RPC.setProtocolEngine()把 ApplicationClientProtocolPB 这个 Protocol 的引擎设置成 ProtobufRpcEngine。这一步并不复杂,只是把配置块 conf 中的配置项“rpc.engine.ApplicationClientProtocolPB”动态设置成 ProtobufRpcEngine 而已。

然后通过 RPC.getProxy()创建 ApplicationClientProtocolPB 的 proxy,这我们在前面已经看见过了。

总之,前面的 proxy.submitApplication(),实际上就是由 protoc 编译生成的 ApplicationClientProtocolService.BlockingInterface.submitApplication()。

还要强调,这个 proxy 存在于用户为提交运行具体应用而起的那个 JVM 上,它既不属于 ResourceManager,也不属于 NodeManager,而是一个独立的 Java 虚拟机,可以是在集群内的任何一台机器上。以示例 WordCount 为例,使用者在某台机器上键入命令行“yarn WordCount”或直接就键入“java WordCount”,就在这台机器上启动了一个 JVM 来执行 WordCount.class。

通过 proxy 发出的 SubmitApplicationRequest,是以 RM 节点为目标的,最终经由操作系统提供的网络传输层以 TCP 报文的方式送达 RM 所在节点机上的对等层,那上面是 ProtoBuf,它会从 TCP 报文中还原出对端所发送的对象。再往上,那就是同样也实现了 ApplicationClientProtocolPB 界面的 ApplicationClientProtocolPBServiceImpl,ProtoBuf 这一层会根据对方请求直接就调用其 submitApplication()。

这样,Client 一侧对于 ApplicationClientProtocolPBClientImpl 所提供函数的调用就转化成 Server 一侧对于 ApplicationClientProtocolPBServiceImpl 所提供的对应函数的调用。当然,Server 一侧函数调用的返回值也会转化成 Client 一侧的返回值,这就实现了远程过程调用 RPC。不言而喻,Client/Server 双方的这两个对象必须提供对同一个界面的实现,在这里就是 ApplicationClientProtocolPB。

这里还要说明一个问题。从软件结构的角度看,我们常把 Client 这一边实际提供服务的这一层看成最高层,它的下面是 ProtoBuf,再下面才是网络传输层,而且函数调用的层次也遵循着同样的顺序,即:

```
YARNRunner.submitJob() //这是处于顶层的应用层
ResourceMgrDelegate.submitApplication() //这是 RM 的代理
YarnClientImpl.submitApplication() //YARN 框架的 Client 一侧
ApplicationClientProtocolPBClientImpl.submitApplication()
//ApplicationClientProtocol 界面
proxy.submitApplication() //ApplicationClientProtocolPB 界面
```

```
Protocol 内部实现的 submitApplication() //在 TCP/IP 的基础上发送应用层的请求
Socket 和 TCP/IP //这是网络连接的最低层
```

可是 Server 这一边就不同了。在 Server 这一边,结构的层次和函数调用的层次是相反的,结构上处于最底层的 Socket 和 TCP/IP 反倒处于函数调用栈的最高层,愈往下调用实质上就愈往结构上的高层走。这是因为 TCP/IP 报文最初到达的是底层,然后逐层往上递交的过程一般都是通过函数调用实现的,所以层层往下调用的过程反倒变成了层层往上递交的过程。

不过我们倒也没有必要从 Socket 这一层开始,我们也可以跳过 ProtoBuf 这一层,因为前面已经讲过了,直接就从 ApplicationClientProtocolPBServiceImpl.submitApplication() 开始。这里先看一下 ApplicationClientProtocolPBServiceImpl 的摘要:

```
class ApplicationClientProtocolPBServiceImpl implements ApplicationClientProtocolPB{
} ApplicationClientProtocol real
} ApplicationClientProtocolPBServiceImpl(ApplicationClientProtocol impl)
> this.real = impl
} getApplicationReport(RpcController arg0, GetApplicationReportRequestProto proto)
> GetApplicationReportRequestPBImp request =
    new GetApplicationReportRequestPBImp(proto)
> GetApplicationReportResponse response = real.getApplicationReport(request)
> return ((GetApplicationReportResponsePBImp)response).getProto()
} getClusterMetrics(RpcController arg0, GetClusterMetricsRequestProto proto)
> GetClusterMetricsRequestPBImp request = new GetClusterMetricsRequestPBImp(proto)
> GetClusterMetricsResponse response = real.getClusterMetrics(request)
> return ((GetClusterMetricsResponsePBImp)response).getProto()
} submitApplication(RpcController arg0, SubmitApplicationRequestProto proto)
> SubmitApplicationRequestPBImp request = new SubmitApplicationRequestPBImp(proto)
> SubmitApplicationResponse response = real.submitApplication(request)
> return ((SubmitApplicationResponsePBImp)response).getProto()
```

显然,这个对象内部有个成分 real,这是个实现了 ApplicationClientProtocol 界面的某类对象,而 ApplicationClientProtocolPBServiceImpl 实现的则是 ApplicationClientProtocolPB 界面。与 Client 侧比较一下,可知那里实现着 ApplicationClientProtocol 界面的就是 ApplicationClientProtocolPBClientImpl,实现 ApplicationClientProtocolPB 界面的则是 proxy 所属的无名类,那是在 ProtoBuf 层的代码中动态定义的,论结构层次前者在上而后者在下。而调用层次就反过来了,ApplicationClientProtocolPBServiceImpl 在上而 real 所属的类在下。所以 real 所属的类所在的层次与 ApplicationClientProtocolPBClientImpl 对等。

这里所看到的所有方法都是通过由 real 所提供的同名方法完成的,而且都遵循着相同的三步操作模式:先准备一个为具体操作而定义请求 request;然后以此为参数调用 real 所提供的某个方法,获取所返回的响应 response;最后从这 response 中提取信息,生成上一层规程(proto)所要求的响应报文。

那么这个 real 对象究竟是什么呢? 这里我们只知道,它属于某个实现了 ApplicationClientProtocol

界面的类,因为 `ApplicationClientProtocol` 是个界面而不是类。并且从代码中可以看出,这个对象是在创建 `ApplicationClientProtocolPBServiceImpl` 对象时作为参数传下来的。现在的问题是,是谁创建了 `ApplicationClientProtocolPBServiceImpl`?

这个问题要从 `ClientRMService` 讲起。`ClientRMService` 是 RM 这一边专门为 Client(而不是 `NodeManager`)服务的,其作用就像是 YARN 子系统为 Client 即平台使用者提供服务的窗口,接受用户提交的作业请求就是其最主要的职能。RM 在其初始化过程 `serviceInit()` 中调用 `createClientRMService()` 创建了一个 `ClientRMService` 对象。下面是这个类的摘要:

```
class ClientRMService extends AbstractService implements ApplicationClientProtocol {}
] YarnScheduler scheduler          //这是 RM 中的作业管理者
] RMContext rmContext              //记录着有关 RM 的种种基本情况
] RMAppManager rmAppManager       //这是 RM 中的 App 管理者
] Server server                    //作为其基础的 RPC 层 server
] ClientRMService(RMContext rmContext, YarnScheduler scheduler, ...)
] serviceInit(Configuration conf)
    > clientBindAddress = getBindAddress(conf)
    > super.serviceInit(conf)
] serviceStart()
    > conf = getConfig()
    > YarnRPC rpc = YarnRPC.create(conf)    //创建一个 YarnRPC 对象
    > this.server = rpc.getServer(ApplicationClientProtocol.class, this, clientBindAddress, ...)
        == HadoopYarnProtoRPC.getServer(Class protocol, Object instance, ...)
        //创建作为 ClientRMService 基础的 RPC 层 server
    >> RpcServerFactoryPBImpl factory = RpcFactoryProvider.getServerFactory(conf)
    >> return RpcServerFactoryPBImpl.getServer(protocol, instance, addr, conf, ...)
    > this.server.start()
    > clientBindAddress = conf.updateConnectAddr(YarnConfiguration.RM_BIND_HOST,
        YarnConfiguration.RM_ADDRESS, ...)
    > super.serviceStart()
] checkAccess(UserGroupInformation callerUGI, String owner, ...)
] submitApplication(SubmitApplicationRequest request)
] ... //定义于 ApplicationClientProtocol 界面的其他方法
```

可以看到, `submitApplication()` 就是它提供的方法之一。更重要的是,从 `serviceStart()` 的摘要可见,此种服务是建立在 RPC 的基础之上的,所以前提就是得创建一个 RPC 层的 Server。但是,同样属于 RPC 层,因其面向和处理的业务不同,所定义的规程(Protocol)也就不同,从而具体 Server 的许多细节也就不同。那么,这里要创建的是什么样的 Server 呢? 我们看到这里以 `ApplicationClientProtocol.class` 为参数调用 `rpc.getServer()`,意思是要创建实现着 `ApplicationClientProtocol` 界面的 Server。值得一提的是,在 Hadoop 的代码中有两个 `ApplicationClientProtocol`: 一个是 class, 那是 `protoc` 在编译 `ProtocolBuf` 定义文件 `applicationclient_protocol.proto` 时自动生成的;另一个是 interface, 那是 Hadoop 本身的代码

中就有的。虽然二者同名,但所属的 package 不同,而源文件 ClientRMService.java 中所 import 的是“... yarn.api.ApplicationClientProtocol”,所以是 interface ApplicationClientProtocol。

另外,这里 rpc 的类型是 YarnRPC,但那只是个抽象类,所以实际上是扩充、落实了 YarnRPC 的 HadoopYarnProtoRPC。而 HadoopYarnProtoRPC.getServer() 则在前面的摘要中已予展开,是通过 RpcServerFactoryPBImpl.getServer() 获取或创建这个 RPC 层 Server。

所以我们接着看 RpcServerFactoryPBImpl.getServer() 的摘要,这里有名堂了:

```
[ResourceManager.serviceInit() > createClientRMService() > ClientRMService.serviceStart() >
HadoopYarnProtoRPC.getServer() > RpcServerFactoryPBImpl.getServer()]

RpcServerFactoryPBImpl.getServer(Class<?> protocol, Object instance,
    InetAddress addr, Configuration conf,
    SecretManager<? extends TokenIdentifier> secretManager,
    int numHandlers, String portRangeConfig)
> Constructor<?> constructor = serviceCache.get(protocol)
    //先在 serviceCache 中寻找,看有没有实现这 protocol 界面的类的构造函数
> if (constructor == null) {      //如果还没有
>+ String name = getPbServiceImplClassName(protocol)
    //获取实现这个界面的类的名称,具体获取的方法见下:
>+> srcPackagePart = getPackageName(clazz)
    //clazz 就是 protocol,即 ApplicationClientProtocol.class
    // srcPackagePart == "org.apache.hadoop.yarn.api"
>+> srcClassName = getClassName(clazz) // "ApplicationClientProtocol"
>+> destPackagePart = srcPackagePart + "." + PB_IMPL_PACKAGE_SUFFIX
    // destPackagePart == "org.apache.hadoop.yarn.api.impl.pb.service"
>+> destClassPart = srcClassName + PB_IMPL_CLASS_SUFFIX // "PBServiceImpl"
>+> return destPackagePart + "." + destClassPart
    // "org.apache.hadoop.yarn.api.impl.pb.service.ApplicationClientProtocolPBServiceImpl"
    //现在有了实现这个界面的类的完整路径
>+ Class<?> pbServiceImplClazz = localConf.getClassByName(name) //获取其 Class 对象
>+ constructor = pbServiceImplClazz.getConstructor(protocol) //获取这个类的构造函数
    // constructor = ApplicationClientProtocolPBServiceImpl()
>+ constructor.setAccessible(true)
>+ serviceCache.putIfAbsent(protocol, constructor) //缓存在 serviceCache,以后就简单了
> }
> Object service = constructor.newInstance(instance)
    //创建 ApplicationClientProtocolPBServiceImpl 对象
> Class<?> pbProtocol = service.getClass().getInterfaces()[0]
    //这个 Interface 就是 ApplicationClientProtocolPB
> Method method = protoCache.get(protocol) //仍是 ApplicationClientProtocol
    //看 protoCache 中有没有 Protoc 为此界面生成的函数 newReflectiveBlockingService()
```

```

> if (method == null) { //没有
>+ Class<?> protoClazz = localConf.getClassByName(getProtoClassName(protocol))
>+ method = protoClazz.getMethod("newReflectiveBlockingService",
                                   pbProtocol.getInterfaces()[0])
>+ method.setAccessible(true)
>+ protoCache.putIfAbsent(protocol, method) //将其缓存在 protoCache 中
> }
> return createServer (pbProtocol, addr, conf, secretManager, numHandlers,
                       (BlockingService)method.invoke(null, service),
                       portRangeConfig)
                       //创建 ApplicationClientProtocolPBServiceImpl 的 RPC 层 Server

```

先看调用参数。这里的参数 protocol 是 ApplicationClientProtocol.class, instance 是个 ClientRMService 对象,我们将会看到,正是这个对象将成为前面所说的 real。

既然形式参数 protocol 实际上是 ApplicationClientProtocol.class,从中就可以提取其类型名称等信息。在此基础上,这里生成一个中间层即 ApplicationClientProtocolPBServiceImpl 的类名,并获取其构造函数,以创建一个此类对象。但是这个对象还不是 RPC 层的 Server, RPC 层 Server 其实是由 protoc 自动生成的一个 BlockingService。所以这里又进一步通过 createServer() 创建一个 BlockingService 对象作为 RPC 层 Server。表面上这里并未把对于中间层 ApplicationClientProtocolPBServiceImpl 的引用保存在一个变量中,但是实际上这已经深埋在 RPC 层 Server 内部,因为这里的变量 service 是对 newReflectiveBlockingService() 做反射式调用(运行时,而不是编译时,按方法名找到这个函数,再加以调用)时的参数。

这样,以后每当 RPC 层的 Server 接收到一个定义于 ApplicationClientProtocol 界面的 RPC 请求时,就都会以此请求为参数调用 ApplicationClientProtocolPBServiceImpl 中的相应方法。对于作业提交,这就到了 ApplicationClientProtocolPBServiceImpl.submitApplication()。

回看前面 ApplicationClientProtocolPBServiceImpl 中对于 real.submitApplication() 的调用,现在我们知道了,real 就是 ClientRMService,所以这就是 ClientRMService.submitApplication()。我们继续往下看。

```

[ApplicationClientProtocolPBServiceImpl.submitApplication()
> ClientRMService.submitApplication()]

public SubmitApplicationResponse submitApplication(SubmitApplicationRequest request)
                                                    throws YarnException {

    ApplicationSubmissionContext submissionContext =
                                                request.getApplicationSubmissionContext();
    ApplicationId applicationId = submissionContext.getApplicationId();

    // ApplicationSubmissionContext needs to be validated for safety - only those fields
    // that are independent of the RM's configuration will be checked here, those that are
    // dependent on RM configuration are validated in RMAppManager.

```



```

String user = null;
try {
    // Safety
    user = UserGroupInformation.getCurrentUser().getShortUserName();
} catch (IOException ie) {
    LOG.warn("Unable to get the current user.", ie); //不能获取用户名
    RMAuditLogger.logFailure(user, AuditConstants.SUBMIT_APP_REQUEST,
        ie.getMessage(), "ClientRMService", "Exception in submitting application", applicationId);
    throw RPCUtil.getRemoteException(ie);
}

// Check whether app has already been put into rmContext
// If it is, simply return the response
if (rmContext.getRMApps().get(applicationId) != null) { //App 已在队列中,是重复提交
    LOG.info("This is an earlier submitted application: " + applicationId);
    return SubmitApplicationResponse.newInstance();
}

if (submissionContext.getQueue() == null) { //如果未指定提交到哪个队列
    submissionContext.setQueue(YarnConfiguration.DEFAULT_QUEUE_NAME);
}

if (submissionContext.getApplicationName() == null) { //如果未提供 App 名称
    submissionContext.setApplicationName(
        YarnConfiguration.DEFAULT_APPLICATION_NAME);
}

if (submissionContext.getApplicationType() == null) { //如果未提供 App 类型
    submissionContext.setApplicationType(
        YarnConfiguration.DEFAULT_APPLICATION_TYPE);
} else { //须检查提供的 App 类型名称是否太长
    if (submissionContext.getApplicationType().length() >
        YarnConfiguration.APPLICATION_TYPE_LENGTH) {
        submissionContext.setApplicationType(submissionContext.getApplicationType()
            .substring(0, YarnConfiguration.APPLICATION_TYPE_LENGTH));
    }
}

try {
    // call RMAppManager to submit application directly
    rmAppManager.submitApplication(submissionContext,
        System.currentTimeMillis(), user); //把 App 交到了 RMAppManager 的手里
    LOG.info("Application with id " + applicationId.getId() + " submitted by user " + user);
    RMAuditLogger.logSuccess(user, AuditConstants.SUBMIT_APP_REQUEST,

```

```
        "ClientRMService", applicationId);  
    } catch (YarnException e) {  
        LOG.info("Exception in submitting application with id " + applicationId.getId(), e);  
        RMAuditLogger.logFailure(user, AuditConstants.SUBMIT_APP_REQUEST,  
            e.getMessage(), "ClientRMService", "Exception in submitting application", applicationId);  
        throw e;  
    }  
    SubmitApplicationResponse response = recordFactory.newRecordInstance(  
        SubmitApplicationResponse.class); //创建一个 SubmitApplicationResponse  
    return response;  
}
```

这里的 `rmAppManager` 是个 `RMAppManager` 类的对象。`RMAppManager` 类对象相当于“中央”的一个部门。`ResourceManager` 要管的事多得很,对于 `App`(作业)的管理只是其中之一,而 `RMAppManager` 就是专门管这个事的。如果不考虑容错所需的备份,那么整个 Hadoop 系统中只有一个 `ResourceManager`,也只有一个 `RMAppManager`,而且就是由 `ResourceManager` 所创建的。可见,当一个作业,也就是一个 `App`,被提交到“中央”的时候,是被交到了 `RMAppManager` 对象的手里。从作业提交的角度看,一旦进入了 `RM` 节点上的 `RMAppManager.submitApplication()`,作业的提交就已完成。至于这以后的处理,那是 `RM` 的事了。

第 6 章

作业的调度与指派

6.1 作业的受理

我们在上一章的结尾处看到,当一个作业,也就是一个 App,通过 RPC 被提交到 RM 节点时,“上岸”后经由 ClientRMService 被交到了 RMApplManager 对象的手里。RM 节点上的 ClientRMService 对象相当于接待站,而 RMApplManager 对象则专门管理与应用/作业的申请和运行有关的事务。两个对象均由 ResourceManager 创建,都在同一个 JVM 上。

ClientRMService 是通过调用 rmApplManager.submitApplication() 把作业申请交到 RMApplManager 手里的,我们就从这里继续往下看。

```
[RPC > ApplicationClientProtocolPBServiceImpl.submitApplication() >
ClientRMService.submitApplication() > RMApplManager.submitApplication()]
```

```
RMApplManager.submitApplication (ApplicationSubmissionContext submissionContext,
                                long submitTime, String user) throws YarnException {
    ApplicationId applicationId = submissionContext.getApplicationId();

    RMApplImpl application =
        createAndPopulateNewRMAppl(submissionContext, submitTime, user);
    ApplicationId appId = submissionContext.getApplicationId();

    if (UserGroupInformation.isSecurityEnabled()) { //如果开启了安全机制
        try {
            this.rmContext.getDelegationTokenRenewer().addApplicationAsync(appId,
                                    parseCredentials(submissionContext),
                                    submissionContext.getCancelTokensWhenComplete(),
                                    application.getUser());
        } catch (Exception e) { //异常,addApplicationAsync()失败
            LOG.warn("Unable to parse credentials.", e);
            // Sending APP_REJECTED is fine, since we assume that the
            // RMAppl is in NEW state and thus we haven't yet informed the
            // scheduler about the existence of the application
        }
    }
}
```

```

        assert application.getState() == RMApState.NEW;
        this.rmContext.getDispatcher().getEventHandler()
            .handle(new RMApRejectedEvent(applicationId, e.getMessage()));
        throw RPCUtil.getRemoteException(e);
    }
} else { //未开启安全机制
    // Dispatcher is not yet started at this time, so these START events
    // enqueued should be guaranteed to be first processed when dispatcher
    // gets started
    this.rmContext.getDispatcher().getEventHandler()
        .handle(new RMApEvent(applicationId, RMApEventType.START));
    //从 rmContext 获取所用的 Dispatcher,进而获取其 EventHandler
    //将 START 事件交给它处理。后者会用其触发 RMApImpl 对象的状态机
}
}
}

```

首先是 `createAndPopulateNewRMAp()`。顾名思义这个函数做的是两件事,即为所提交的作业创建一个 `RMAp` 对象,并 `Populate`。不过 `RMAp` 只是个 `interface`,实际创建的是个 `RMApImpl` 对象。这是根据所提交的 `submissionContext`,即前一章中所讲的 `ASC` 和用户名创建的,代表着具体的 `App`,也包含着有关该 `App` 的全部信息。至于 `Populate`,则是说把所创建的 `RMApImpl` 对象连同其 `ApplicationId` 放到一个类似于名单一样的便查表中,即 `MAP<ApplicationId, RMAp>` 中,这样以后凭 `ID` 就可以从中找到它的 `RMApImpl` 对象。

```

[RPC > ApplicationClientProtocolPBServiceImpl.submitApplication() >
ClientRMService.submitApplication() > RMApManager.submitApplication() >
createAndPopulateNewRMAp()]

```

```

createAndPopulateNewRMAp(ApplicationSubmissionContext submissionContext,
                           long submitTime, String user)
> ApplicationId applicationId = submissionContext.getApplicationId()
> ResourceRequest amReq = validateAndCreateResourceRequest(submissionContext)
    //检验资源请求是否合理,不合理就发 InvalidResourceRequestException 异常
> RMApImpl application = new RMApImpl(applicationId, rmContext,
    this.conf, submissionContext.getApplicationName(), user,
    submissionContext.getQueue(), submissionContext,
    this.scheduler, this.masterService, submitTime,
    submissionContext.getApplicationType(), //默认为“YARN”
    submissionContext.getApplicationTags(), amReq)
> rmContext.getRMAps().putIfAbsent(applicationId, application)
    //将 App 加入 RMContextImpl 内部的一个 MAP
    //以后凭 applicationId 就可找到其 RMApImpl 对象

```

```

> this.applicationACLsManager.addApplication(applicationId,
    submissionContext.getAMContainerSpec().getApplicationACLs())
    //将 App 加入到访问控制名单 ACL 管理器中
> String appViewACLs = submissionContext.getAMContainerSpec()
    .getApplicationACLs().get(ApplicationAccessType.VIEW_APP)
> rmContext.getSystemMetricsPublisher().appACLsUpdated(application,
    appViewACLs, System.currentTimeMillis()) //与统计有关
> return application

```

创建 `RMAppl` 时所传递的参数主要来自用户,特别是来自其提交的 `ApplicationSubmissionContext`,但也有些信息是由 RM 节点上的 `RMApplManager` 提供的,如 `rmContext`、`this.conf`、`this.scheduler` 和 `this.masterService`。其中 `this.scheduler` 决定了将采用哪一种调度器;`this.masterService` 则是个 `ApplicationMasterService` 对象,是 RM 节点上专门管理那些为具体应用而建立的 `ApplicationMaster` 的。如前所述,`ApplicationMaster` 类似于“项目组长”。下面就要为当前的这个 App 找个节点在上面创建 `ApplicationMaster` 了,创建后它就得向 `ApplicationMasterService` 登记报到。

回到前面 `RMApplManager.submitApplication()` 的代码。

Hadoop 实现了基于身份认证的安全机制,但是用户可以启用也可以不启用这种安全机制。启用了安全机制后的流程当然要稍为复杂一些,但是由此而来的复杂性并不影响 Hadoop 与 YARN 的主流。在这里,如果一头钻进安全机制不免会偏离我们的大方向,会冲淡我们对主流的分析理解,所以我们现在走不开启安全机制的这条路。

作业的提交引起一个 `RMApplEventType.START` 事件,这就涉及我们在前面介绍过的状态机了。如前所述,用户每向 RM 提交一个作业,`RMApplManager` 就为其创建一个 `RMApplImpl` 对象,作为该作业在 RM 中的代表。可想而知,随着作业的进展,其 `RMApplImpl` 对象将处于各种不同的状态。所以,每个 `RMApplImpl` 对象,或者说每个 `RMAppl`,都有个自己的状态机。所有 `RMApplImpl` 对象的状态机都是一样的,只不过各自处于不同的状态;所以 `stateMachineFactory` 是 `RMApplImpl` 类的静态成分。但是,具体的状态机 `stateMachine` 则是在 `RMApplImpl` 对象的构造函数中创建的。创建后的初始状态为 `RMApplState.NEW`。而事件 `RMApplEventType.START` 就被用来触发该 App 的状态机。

不过,这里也简单提一下启用安全机制时的流程。代码中的 `getDelegationTokenRenewer()` 返回一个 `DelegationTokenRenewer` 对象,这个对象提供一个方法 `addApplicationAsync()`,这显然就是异步增加一个 App 的意思。这个方法将 App 连同用户信息及其各种证件一起挂入 `DelegationTokenRenewer` 的队列,然后它的一个 `DelegationTokenRenewerRunnable` 线程会来检验队列中这些 App 的资质。如果检验合格,就同样会发起一次 `RMApplEventType.START` 事件,此后就跟不开启安全机制时的流程一样了。

这里创建的事件是 `RMApplEventType.START`,并通过 `Dispatcher` 把这个事件派送给 `RMAppl` 的状态机。前面讲状态机时说过,`Dispatcher` 就好像是按事件类型导向的路由器,想要让某个受主接受某类事件,就得向 `Dispatcher` 登记,这就好比在路由器中配置一条路由。实际情况也正是如此。`ResourceManager` 在其 `serviceInit()` 中向其内部的 `rmDispatcher` 登记,将 `RMApplEventType` 类的事件绑定到一个 `ApplicationEventDispatcher` 类对象,并将后者记

录在 `rmContext` 中。注意,后者虽然名为 `ApplicationEventDispatcher`,但实际上却只是 `RMAp` 的 `EventHandler`。所以,上面的 `this.rmContext.getDispatcher().getEventHandler()` 所返回的其实是作为事件处理器的 `ApplicationEventDispatcher` 对象,然后调用其 `handle()` 方法就会触发 `RMAp` 状态机的跳变:

```
[RPC > ApplicationClientProtocolPBServiceImpl.submitApplication() >
ClientRMService.submitApplication() > RMApManager.submitApplication() >
ApplicationEventDispatcher.handle()]
```

```
ApplicationEventDispatcher.handle(RMApEvent event)
```

```
> appID = event.getApplicationId() //获取该事件的 AppId
```

```
> rmApp = this.rmContext.getRMAps().get(appID) //根据 AppId 找到其 RMApImpl 对象
```

```
> if (rmApp != null) rmApp.handle(event) == RMApImpl.handle(event) //调用其 handle() 方法
```

```
>> appID = event.getApplicationId()
```

```
>> this.stateMachine.doTransition(event.getType(), event) //触发状态机跳变
```

这里 `rmContext` 是 `RM` 的 `Context`,实际上是个 `RMContextImpl` 对象。前述的那个 `MAP` 就在这个对象内部,所以根据具体应用的 `AppID` 可以得到其 `RMAp`,实际上是代表着这个应用的 `RMApImpl` 对象。

看一下 `RMApImpl` 类的代码,就可以找到其状态机的跳变规则,其中以 `RMApEventType.START` 为触发条件的只有一条,那就是:

```
addTransition(RMApState.NEW, RMApState.NEW_SAVING,
```

```
    RMApEventType.START, new RMApNewlySavingTransition())
```

我们这个应用的 `RMApImpl` 对象是刚刚创建的,它的状态机正处于初始状态 `RMApState.NEW`,所以这条规则正好适用。注意,这条规则还规定要为状态机提供一个 `RMApNewlySavingTransition` 类的对象,这个类提供了一个 `transition()` 函数供状态机调用,作为这个跳变所需的伴随操作,这也是状态机对事件的反应之一。类似于 `RMApNewlySavingTransition` 那样,用在这个地方的类,都实现了 `SingleArcTransition` 或 `MultipleArcTransition` 界面,所以都提供 `transition()` 操作。值得注意的是,名曰 `transition()`,实际上却不是用来实现状态变迁的,而只是用来提供伴随着状态变迁、除状态跳变本身以外的操作。

所以,接着发生的事,是 `RMApNewlySavingTransition.transition()` 得到调用,而这个状态机的当前状态则变成 `RMApState.NEW_SAVING`。

```
[RMApManager.submitApplication() => RMApNewlySavingTransition.transition()]
```

```
RMApNewlySavingTransition.transition(RMApImpl app, RMApEvent event) {
```

```
    // If recovery is enabled then store the application information in a
```

```
    // non - blocking call so make sure that RM has stored the information
```

```
    // needed to restart the AM after RM restart without further client communication
```

```
    LOG.info("Storing application with id " + app.applicationId);
```



```
app.rmContext.getStateStore().storeNewApplication(app);
```

这里在程序的调用路径中用“=>”表示前者导致了后者，而不是（或不一定是）前者直接调用了后者。这是因为，如果 Dispatcher 是异步的，那么前者只是把一个事件对象放进一个事件队列，而另有一个线程在处理这些事件对象的派发，这才调用到了后者，后者实际上并非由前者直接调用，而且后者是在另一个线程的上下文中得到执行。

显然，这个函数要做的事情是通过 storeNewApplication() 把具体应用的信息都保存起来。这是因为，这个应用是否能马上被调度运行还不得而知，也许需要等上很长时间，而且随后对这个应用的处理和执行有可能中途失败，将要为其创建的 AM 甚至 RM 都有可能重启，那时候这个 App 应该要有个副本作为恢复起点，这相当于数据库处理中的 checkpoint。我们在前面看到所创建的 RMApplImpl 对象被放进了 RM 的 app 集合，但是那个正本的 RMApplImpl 对象在随后的处理中是会改变，甚至有可能丢失的。

```
[RMApplManager.submitApplication() => RMApplNewlySavingTransition.transition() >
RMStateStore.storeNewApplication()]
```

```
/* *
```

```
* Non - Blocking API
```

```
* ResourceManager services use this to store the application's state
```

```
* This does not block the dispatcher threads
```

```
* RMApplStoredEvent will be sent on completion to notify the RMAppl
```

```
*/
```

```
storeNewApplication(RMAppl app) {
```

```
    ApplicationSubmissionContext context = app.getApplicationSubmissionContext();
```

```
    assert context instanceof ApplicationSubmissionContextPBImpl;
```

```
    ApplicationState appState = new ApplicationState(app.getSubmitTime(),
```

```
                                                    app.getStartTime(), context, app.getUser());
```

```
    dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState));
```

```
}
```

显然，这里要保存的主体是这个 App 的 ApplicationSubmissionContext，用户所提交的信息其实都在这里。把它保存起来，就把所提交的作业信息保存了一份副本，所以这里创建的是一个包含着这些信息的 ApplicationState 对象。此外，这里还以此为参数创建了一个 RMStateStoreAppEvent 事件，并让 Dispatcher 把该事件分发给相应的事件处理器。不过，这个事件的类型属于 RMStateStoreEvent，而不是前述的 RMApplState，所以针对的是不同的事件处理器或状态机。事实上，这一方面固然是为了保存所创建的 ApplicationState 对象 appState，另一方面这也会引起 RMStateStore 状态机的跳变和反应。这个 RMStateStoreAppEvent 的具体类型是 RMStateStoreEventType.STORE_APP，appState 则相当于这个事件的附件。

那保存在哪里呢？RM 节点上有个 RMStateStore 对象，我们可以在 ResourceManager 的

serviceInit()中看到 RMStateStore 的创建。但 RMStateStore 是个抽象类,由此扩充派生出来的就有 FileSystemRMStateStore、MemoryRMStateStore、NullRMStateStore 以及 ZKRMStateStore,具体选用哪一种可以在配置文件 yarn-default.xml 中加以指定。

可想而知,如果采用的是 FileSystemRMStateStore,那就要把副本保存在文件系统中,这里面有个过程,而且可能要应对许多不同的情况。所以,其实 RMStateStore 类自己就有个 StateMachineFactory,具体的 FileSystemRMStateStore 对象则各有自己的状态机。

RMStateStore 对 STORE_APP 事件的反应是在 StoreAppTransition.transition()中调用 storeApplicationStateInternal()以保存 appState 中的数据,操作完成以后就发出一个 RMAppEventType.APP_NEW_SAVED 事件,用来触发 RMAppImpl 对象的状态机。

在 RMAppImpl 对象的状态机中,以 RMAppEventType.APP_NEW_SAVED 为触发条件的跳变规则是:

```
addTransition(RMAppState.NEW_SAVING, RMAppState.SUBMITTED,
    RMAppEventType.APP_NEW_SAVED, new AddApplicationToSchedulerTransition())
```

就是说,如果状态机的当前状态是 NEW_SAVING,到来的事件是 APP_NEW_SAVED,则状态变成 SUBMITTED,伴随的操作是 AddApplicationToSchedulerTransition.transition()。状态变成 RMAppState.SUBMITTED,就表示提交已经完成。不过状态的变化发生在伴随操作之后,所以先要执行下面这个操作:

```
[RMAppManager.submitApplication() => RMAppNewlySavingTransition.transition() =>
    AddApplicationToSchedulerTransition.transition()]
```

```
AddApplicationToSchedulerTransition.transition(RMAppImpl app, RMAppEvent event) {
    app.handler.handle(new AppAddedSchedulerEvent(app.applicationId,
        app.submissionContext.getQueue(), app.user,
        app.submissionContext.getReservationID()));
}
```

这里创建并作为参数传给 app.handler.handle()的 AppAddedSchedulerEvent 对象把有关该 app 的一些信息打包在一起,其中 applicationId 和用户名 user 不言自明;而 getQueue()所返回的队列名表示要把该 app 请求挂到哪一个资源调度队列中。最后的 ReservationID 则用于事先已有预订的情况。我们来看一下 AppAddedSchedulerEvent 对象的构造函数:

```
class AppAddedSchedulerEvent extends SchedulerEvent {
    public AppAddedSchedulerEvent(ApplicationId applicationId, String queue,
        String user, boolean isAppRecovering, ReservationId reservationID) {
        super(SchedulerEventType.APP_ADDED);
        this.applicationId = applicationId;
        this.queue = queue;
        this.user = user;
        this.reservationID = reservationID;
        this.isAppRecovering = isAppRecovering;
    }
}
```

```

    }
}

```

这里我们关心的是,这个事件的类型是 `SchedulerEventType.APP_ADDED`。

另外,这个事件是通过 `app.handler.handle()` 派发的;而 `RMAAppImpl` 的 `handler`,到源码中看一下就可知道,是通过 `dispatcher.getEventHandler()` 得来的。那么这个 `EventHandler` 究竟是什么呢?这就要看当初登记与 `SchedulerEventType` 绑定的是什么。事实上,前面我们看到在 `ResourceManager` 的初始化阶段,在其所创建的 `RMAActiveServices` 的 `serviceInit()` 中创建了一个 `SchedulerEventDispatcher` 对象,并登记与 `SchedulerEventType` 类的事件绑定,这个 `SchedulerEventDispatcher` 实现了 `EventHandler` 界面,起着 `EventHandler` 的作用。所以这里调用的实际上是 `SchedulerEventDispatcher.handle(SchedulerEventType.APP_ADDED)`。

注意 `SchedulerEventDispatcher` 是定义于 `ResourceManager` 内部的一个类,它实现了 `EventHandler` 界面,并且内部有一个 `EventProcessor` 线程。现在调用的是 `EventHandler` 界面上的 `handle()` 方法:

```

[RMAppManager.submitApplication() => RMAAppNewlySavingTransition.transition()
=> AddApplicationTbSchedulerTransition.transition() > SchedulerEventDispatcher.handle()]

ResourceManager.SchedulerEventDispatcher.handle(SchedulerEvent event) {
    try {
        int qSize = eventQueue.size();
        if (qSize != 0 && qSize % 1000 == 0) {
            LOG.info("Size of scheduler event - queue is " + qSize); //每 1000 个事件 LOG 一次
        }
        int remCapacity = eventQueue.remainingCapacity();
        if (remCapacity < 1000) { //事件队列的容量已感紧张
            LOG.info("Very low remaining capacity on scheduler event queue: " + remCapacity);
        }
        this.eventQueue.put(event); //挂入调度器的事件队列
    } catch (InterruptedException e) {
        LOG.info("Interrupted. Trying to exit gracefully.");
    }
}
}

```

像 `AsyncDispatcher` 中的情况一样,这个 `handle()` 只是把作为参数传下来的事件挂在 `SchedulerEventDispatcher` 的事件队列中就返回了。但是 `SchedulerEventDispatcher` 内部的 `EventProcessor` 线程会从这事件队列中取出事件做进一步的处理,它的 `run()` 函数如下:

```

[SchedulerEventDispatcher.EventProcessor.run()]

public void run() {
    SchedulerEvent event;

```

```

while (!stopped && !Thread.currentThread().isInterrupted()) {
    try {
        event = eventQueue.take(); //从队列中取下一个事件
    } catch (InterruptedException e) {
        LOG.error("Returning, interrupted : " + e);
        return; // TODO: Kill RM.
    }

    try {
        scheduler.handle(event); //例如 FifoScheduler.handle()
    } catch (Throwable t) {
        // An error occurred, but we are shutting down anyway
        // If it was an InterruptedException, the very act of
        // shutdown could have caused it and is probably harmless
        if (stopped) {
            LOG.warn("Exception during shutdown: ", t);
            break;
        }
        LOG.fatal("Error in handling event type " + event.getType() + " to the scheduler", t);
        if (shouldExitOnError && !ShutdownHookManager.get().isShutdownInProgress()) {
            LOG.info("Exiting, bbye...");
            System.exit(-1);
        }
    }
} //end catch
} //end while
}

```

SchedulerEventDispatcher 的 scheduler, 就是 ResourceManager 的 scheduler, 如前所述可有三种选择, 这里假定其为 FifoScheduler, 所以这里所调用的就是 FifoScheduler.handle()。我们看一下它的摘要:

```
FifoScheduler.handle(SchedulerEvent event)
```

```

> switch(event.getType()) {
> case NODE_ADDED: //增加了一个节点
>> ...
> case NODE_REMOVED: //撤去了一个节点
>> ...
> case NODE_RESOURCE_UPDATE: //节点的资源发生变化
>> ...
> case NODE_UPDATE: //节点的状态发生变化
>> ...

```

```

> case APP_ADDED:      //增加了一个 App
>> appAddedEvent = (AppAddedSchedulerEvent) event
>> addApplication(appAddedEvent.getApplicationId(), appAddedEvent.getQueue(), ... )
>>> application = new SchedulerApplication<FiCaSchedulerApp>( DEFAULT_QUEUE, user )
>>> applications.put(applicationId, application)
                        //将创建的 SchedulerApplication 放在 applications 集合中
>>> metrics.submitApp(user) //metrics 是一个 QueueMetrics 对象,用于统计目的
>>> e = new RMapEvent(applicationId, RMapEventType.APP_ACCEPTED) //创建新事件
>>> rmContext.getDispatcher().getEventHandler().handle(e) //派发新创建的事件
> case APP_REMOVED:    //撤销了一个 App
>> ...
> case APP_ATTEMPT_ADDED: //增加了一个 App 企图
>> ...
> case APP_ATTEMPT_REMOVED: //撤销了一个 App 企图
>> ...
> case CONTAINER_EXPIRED: //容器使用超时
>> ...
> } //end switch

```

同属于 SchedulerEventType 的事件有不少,这个 handle()方法通过一个 switch 语句分情形加以处理。由此可见,作为实现了 EventHandler 界面的对象,其 handle()函数对于事件的反应并不一定是通过状态机做出的,也可以通过 switch 语句做出反应。我们现在这个事件是 APP_ADDED,暂不关心别的事件。

对于 APP_ADDED,具体的反应是 addApplication()。这里把这个函数展开了:首先是为这个 App 创建一个 SchedulerApplication 类的对象 application,这是具体 App 在调度器中的代表,这个 SchedulerApplication 和 applicationId 一起被放入一个集合 applications 中,以后凭 applicationId 就可以在这个集合中找到其 SchedulerApplication 对象。请注意在创建 application 时的参数 DEFAULT_QUEUE,这说明在 ApplicationSubmissionContext 中指定的队列对于 FifoScheduler 不起作用,FifoScheduler 总是采用其 DEFAULT_QUEUE。

同时,出于统计的目的,这里还以用户名 user 为参数向一个 QueueMetrics 类对象 metrics 调用其 submitApp(),表示这个用户增加了一个 App。

然后就创建一个新的事件 RMapEventType.APP_ACCEPTED,表示这个 App 已经被提交给调度器并为其接受。这个事件既然属于 RMapEventType,事件中又包含着原先的 applicationId,那就是回过头来针对原先那个 RMapImpl 状态机的,相应的跳变规则为:

```

addTransition(RMapState.SUBMITTED, RMapState.ACCEPTED,
    RMapEventType.APP_ACCEPTED, new StartAppAttemptTransition())

```

此时 RMapImpl 状态机对触发事件 RMapEventType.APP_ACCEPTED 的反应是:调用伴随操作 StartAppAttemptTransition.transition(),然后使这个状态机的当前状态变成 RMapState.ACCEPTED。

```
[RManager.submitApplication() => RManagerNewlySavingTransition.transition()
=> AddApplicationToSchedulerTransition.transition() => StartAppAttemptTransition.transition()]
```

```
StartAppAttemptTransition.transition(RManagerImpl app, RManagerEvent event) {
```

```
    app.createAndStartNewAttempt(false); //开始一次启动运行的尝试
```

```
}
```

既然 App 申请已为调度器所接受,下一步就是为此应用(作业)开始一次运行尝试,即 AppAttempt。这里的参数 app 是个 RManagerImpl 对象,所以这里所调用的操作就是 RManagerImpl.createAndStartNewAttempt(),其摘要如下:

```
[RManager.submitApplication() => RManagerNewlySavingTransition.transition() =>
AddApplicationToSchedulerTransition.transition() => StartAppAttemptTransition.transition() >
RManagerImpl.createAndStartNewAttempt()]
```

```
RManagerImpl.createAndStartNewAttempt()
```

```
> createNewAttempt() //创建一个新的 RManagerAttemptImpl,并将其设置成 currentAttempt
```

```
>> appAttemptId = ApplicationAttemptId.newInstance(applicationId, attempts.size() + 1)
```

```
>> attempt = new RManagerAttemptImpl(appAttemptId, rmContext,
```

```
        scheduler, masterService, ...)
```

```
>> attempts.put(appAttemptId, attempt) //将 attempt 连同其 ID 一起放入 attempts 集合
```

```
>> currentAttempt = attempt //将新创建的 attempt 设置成 currentAttempt
```

```
> e = new RManagerStartAttemptEvent(currentAttempt.getAppAttemptId(), ...)
```

```
    //生成一个 RManagerAttemptEventType.START 事件
```

```
>> super(appAttemptId, RManagerAttemptEventType.START)
```

```
> handler.handle(e) //handler 取决于事件类型,在这里是 RManagerAttemptImpl.handle()
```

顺便看一下,作为对照,RManagerImpl.createAndStartNewAttempt()的源码是这样:

```
RManagerImpl.createAndStartNewAttempt(boolean transferStateFromPreviousAttempt) {
    createNewAttempt();
    handler.handle(new RManagerStartAttemptEvent(currentAttempt.getAppAttemptId(),
                                                    transferStateFromPreviousAttempt));
}
```

对于用户提交的每个 App,试图启动其运行的每次尝试(试运行)都称为一次应用尝试,即 RManagerAttempt,由一个 RManagerAttemptImpl 类的对象作为代表。之所以加上前缀 RM,显然是因为这是在 RM 节点上。对于具体 App 的一次尝试是个逻辑上独立又颇为复杂的过程,中间有许多状态变迁,所以 RManagerAttemptImpl 对象中也有状态机。

像 ApplicationId 一样,每个 RManagerAttemptImpl 都有个唯一的 ApplicationAttemptId,所以在创建 RManagerAttemptImpl 之前都要为其分配一个号码,每次递增。

在 RManagerImpl 对象内部有个 attempts 集合,实际上是个 Map<ApplicationAttemptId, RManagerAttempt>,凡属于这个 App 的所有 attempt 都在这个集合中,根据具体 AppAttempt

的 ID 号就可在这个集合中找到其 RMApAttemptImpl 对象。所以,每创建了一个 RMApAttemptImpl 对象之后,就要把它连同其 ID 号放到这个集合中,并使其成为这个 RMApAttemptImpl 的“当前尝试”currentAttempt。

然后就创建一个 RMApAttemptEventType.START 事件并加以派发。如前所述,派发的目标取决于事件类型。事件类型与事件处理器的绑定是事先向 Dispatcher 登记好的,所以这里的 handler.handle() 实际上是 ApplicationAttemptEventDispatcher.handle():

```
[RMApManager.submitApplication()=> RMApNewlySavingTransition.transition()=>
AddApplicationToSchedulerTransition.transition()=> StartAppAttemptTransition.transition()>
RMApAttemptImpl.createAndStartNewAttempt()> ApplicationAttemptEventDispatcher.handle()]

ApplicationAttemptEventDispatcher.handle(RMApAttemptEvent event)
> ApplicationAttemptId appAttemptID = event.getApplicationAttemptId()
> ApplicationId appId = appAttemptID.getApplicationId() //注意,这是 ApplicationId
> rMAp = this.rmContext.getRMAps().get(appAttemptID)
//根据 ApplicationId 获取其 RMApAttemptImpl 对象
> if (rMAp != null) {
>+ RMApAttempt rMApAttempt = rMAp.getRMApAttempt(appAttemptID)
//根据 ApplicationAttemptId 获取其 RMApAttemptImpl 对象
>+ if (rMApAttempt != null) rMApAttempt.handle(event) == RMApAttemptImpl.handle()
>+> appAttemptID = event.getApplicationAttemptId() //从 event 中提取
>+> this.stateMachine.doTransition(event.getType(), event)
//这是具体 RMApAttemptImpl 对象的状态机
> }
```

可见,事件处理器 ApplicationAttemptEventDispatcher 对此事件的处理就是驱动具体 RMApAttemptImpl 对象的状态机,而 RMApAttemptImpl 状态机中相应的跳变规则如下:

```
addTransition(RMApAttemptState.NEW, RMApAttemptState.SUBMITTED,
```

```
RMApAttemptEventType.START, new AttemptStartedTransition())
```

显然,RMApAttemptImpl.handle() 会调用 AttemptStartedTransition.transition():

```
[RMApManager.submitApplication()=> RMApNewlySavingTransition.transition()
=> AddApplicationToSchedulerTransition.transition()=> StartAppAttemptTransition.transition()
> RMApAttemptImpl.createAndStartNewAttempt()=> AttemptStartedTransition.transition()]
```

```
AttemptStartedTransition.transition(RMApAttemptImpl appAttempt,
```

```
RMApAttemptEvent event) {
```

```
boolean transferStateFromPreviousAttempt = false;
```

```
if (event instanceof RMApStartAttemptEvent) {
```

```
transferStateFromPreviousAttempt =
```

```
((RMApStartAttemptEvent) event).getTransferStateFromPreviousAttempt();
```

```

    }

    appAttempt.startTime = System.currentTimeMillis();

    // Register with the ApplicationMasterService
    appAttempt.masterService.registerAppAttempt(appAttempt.applicationAttemptId);
                                //向 ApplicationMasterService 登记一次 RMAppAttempt

    if (UserGroupInformation.isSecurityEnabled()) { //如果启用了安全机制
        appAttempt.clientTokenMasterKey =
            appAttempt.rmContext.getClientToAMTokenSecretManager()
                .createMasterKey(appAttempt.applicationAttemptId);
    }

    // Add the applicationAttempt to the scheduler and inform the scheduler
    // whether to transfer the state from previous attempt.
    appAttempt.eventHandler.handle(new AppAttemptAddedSchedulerEvent(
        appAttempt.applicationAttemptId, transferStateFromPreviousAttempt));
}

```

我们为这个函数做一个摘要,并略作展开,就是这样:

```

AttemptStartedTransition.transition(RMAppAttemptImpl appAttempt, RMAppAttemptEvent event)
> transferStateFromPreviousAttempt = event.getTransferStateFromPreviousAttempt()
> appAttempt.masterService.registerAppAttempt(appAttempt.applicationAttemptId)
> e = new AppAttemptAddedSchedulerEvent(appAttempt.applicationAttemptId, ...)
>> super(SchedulerEventType.APP_ATTEMPT_ADDED)
> appAttempt.eventHandler.handle(e) //这就是 FifoScheduler.handle()

```

先说说参数。这里的参数已经从 RMAppImpl 变成了 RMAppAttemptImpl。前者代表一个作业、一个应用;后者则代表运行这个应用的一次尝试、一次试运行。一个应用不一定能一次运行成功,也许需要运行多次才能完成。这里的这个 RMAppAttemptImpl 对象 appAttempt,就是前面在 createAndStartNewAttempt() 中创建的,这是 RMAppImpl 状态机受事件 RMAppEventType.APP_ACCEPTED 的触发而做出的反应。从现在开始,下面的活动就都是针对 App 的一次具体的运行尝试,而不是针对整个 App 的了。

这里的布尔量 transferStateFromPreviousAttempt 最初是从事件对象 event 里带进来的,目的在于供调度器使用,用以确定在为一个 App 发起一次新的 Attempt 时要不要把先前的 Attempt 的状态转移过来。这是因为,在这种情况下先前的 Attempt 多半已经失败,将其状态转移继承过来很可能可以利用其已经获得的部分成果。

另一个事情,是通过 registerAppAttempt() 向 RM 的 ApplicationMasterService 登记,向其报告新创建 RMAppAttemptImpl 对象的 ApplicationAttemptId。因为一旦为此应用在某个 NodeManager 节点上创建起一个 ApplicationMaster,它就要来与 ApplicationMasterService 联系,所以这里先要登记一下,挂个号备个案。ApplicationMasterService 是专门管理正在集群中运

行的那些 App 的 ApplicationMaster 的。

然后就是创建 `AppAttemptAddedSchedulerEvent` 对象并加以派发。这个事件的类型实际上是 `SchedulerEventType.APP_ATTEMPT_ADDED`。在我们的情景中,因为我们假定采用 `FifoScheduler`,这个事件对象又会被派发给 `FifoScheduler.handle()`。我们在前面见过这个函数的摘要,但是那时我们只关心其 `switch` 语句中的“`case APP_ADDED`”,现在要关心一下“`case APP_ATTEMPT_ADDED`”了:

```
FifoScheduler.handle(SchedulerEvent event)
> switch(event.getType()) {
> case APP_ATTEMPT_ADDED:    //增加了一个 App 运行尝试
>> addApplicationAttempt(appAttemptAddedEvent.getApplicationAttemptId(), ...)
> } //end switch
```

这当然是要把我们的 `AppAttempt` 加入到调度器中。下面是 `addApplicationAttempt()` 的源码。

```
[RMAppManager.submitApplication() => RMAppNewlySavingTransition.transition()
=> AddApplicationToSchedulerTransition.transition() => StartAppAttemptTransition.transition()
=> AttemptStartedTransition.transition() => FifoScheduler.addApplicationAttempt()]

FifoScheduler.addApplicationAttempt(ApplicationAttemptId appAttemptId,
    boolean transferStateFromPreviousAttempt, boolean isAttemptRecovering) {
    SchedulerApplication<FiCaSchedulerApp> application =
        applications.get(appAttemptId.getApplicationId());
    //参考前面 FifoScheduler.handle() 中 case APP_ADDED 下的 addApplication()
    String user = application.getUser();
    // TODO: Fix store
    FiCaSchedulerApp schedulerApp =
        new FiCaSchedulerApp(appAttemptId, user, DEFAULT_QUEUE,
            activeUsersManager, this.rmContext);
    //这个类型既用于 FifoScheduler,又用于 CapacityScheduler,故得此名
    if (transferStateFromPreviousAttempt) {
        schedulerApp.transferStateFromPreviousAttempt(
            application.getCurrentAppAttempt());
    } //如果要转移继承先前尝试的状态,那么所谓先前尝试就是此刻的当前尝试
    application.setCurrentAppAttempt(schedulerApp); //新的尝试变成了当前尝试

    metrics.submitAppAttempt(user);    //将 AppAttempt 纳入统计
    LOG.info("Added Application Attempt " + appAttemptId
        + " to scheduler from user " + application.getUser());
    if (isAttemptRecovering) { //见调用参数,true 表示 Attempt 并非新创
        if (LOG.isDebugEnabled()) {
```

```

LOG.debug(appAttemptId + " is recovering. Skipping notifying ATTEMPT_ADDED");
    }
    } else { //如果是新加入的应用尝试
        mContext.getDispatcher().getEventHandler().handle(
            new RMAppAttemptEvent(appAttemptId,
                RMAppAttemptEventType.ATTEMPT_ADDED));
    }
}
}

```

前面,在将一个 App 提交给调度器的时候(当时的事件是 APP_ADDED),我们为其创建了一个 SchedulerApplication 对象。其实这个 SchedulerApplication 只是个简略的模板名,对于 FifoScheduler 而言更确切的类名是 SchedulerApplication<FiCaSchedulerApp>。前缀“FiCa”说明这个类既用于 FifoScheduler,也用于 CapacityScheduler。创建了这个对象之后,就把它放入了调度器的 applications 集合中。需要的时候,用 ApplicationId 就可从这个集合中找到这个 SchedulerApplication 对象。这里的第一个语句正是这样。之所以需要这个对象,是因为这里要为其创建一个专供调度器使用的 FiCaSchedulerApp 对象,这是对 SchedulerApplicationAttempt 的扩展,实际上代表的是应用尝试(而不是应用本身),这个尝试应该成为 App 的“当前尝试(CurrentAppAttempt)”。而如果要转移继承先前尝试的状态,则此刻之前的当前尝试就是此后的先前尝试了。

注意,这个函数有个调用参数 isAttemptRecovering,这是个布尔量,为“真”即表示此次调用 addApplicationAttempt()并非要将一个新创的 Attempt 加入调度,而只是要恢复原来已经存在的 Attempt。

一般而言,调用 addApplicationAttempt()总是为新创的应用尝试居多,此时就得创建和派发 RMAppAttemptEventType.ATTEMPT_ADDED 事件。其实这就是调度器对于具体 RMAppAttemptImpl 对象的响应。那个 RMAppAttemptImpl 状态机的状态在前面已经变成 SUBMITTED,现在又受到 ATTEMPT_ADDED 事件的触发,其跳变规则为:

```

addTransition(RMAppAttemptState.SUBMITTED,
    EnumSet.of(RMAppAttemptState.LAUNCHED_UNMANAGED_SAVING,
        RMAppAttemptState.SCHEDULED),
    RMAppAttemptEventType.ATTEMPT_ADDED, new ScheduleTransition())

```

这条规则与我们以前所见有些不同,它的跳变目标有两个,一个是 SCHEDULED,另一个是 LAUNCHED_UNMANAGED_SAVING。这说明,在同一事件的触发下,状态机的下一个状态是什么还得看某些别的条件才能确定,这就是所谓“多弧(MultipleArc)”跳变。状态机的宿主即其所属对象的 handle()函数对于单弧和多弧的处理有所不同,但是不管是单弧还是多弧,最后的状态改变总是发生在从 transition()函数返回的时候,而且 transition()的返回值决定了多弧跳变的目标状态究竟是这个集合中的哪一种。

我们看一下 ScheduleTransition.transition()的代码:

```

[RMAppManager.submitApplication()=> RMAppNewlySavingTransition.transition()
=> AddApplicationToSchedulerTransition.transition()=> StartAppAttemptTransition.transition()]

```

```
=> AttemptStartedTransition.transition() => FifoScheduler.addApplicationAttempt()
=> RMAppAttemptImpl.ScheduleTransition.transition()]
```

```
ScheduleTransition.transition(RMAppAttemptImpl appAttempt, RMAppAttemptEvent event) {
    ApplicationSubmissionContext subCtx = appAttempt.submissionContext;
    if (!subCtx.getUnmanagedAM()) { //如果不是 UnmanagedAM,就得为其创建 AM
        // Need reset # containers before create new attempt, because this request
        // will be passed to scheduler, and scheduler will deduct the number after
        // AM container allocated

        // Currently, following fields are all hard code,
        // TODO: change these fields when we want to support
        // priority/resource - name/relax - locality specification for AM containers
        // allocation.
        appAttempt.amReq.setNumContainers(1); //只要求一个容器,用于创建 AM
        appAttempt.amReq.setPriority(AM_CONTAINER_PRIORITY);
        appAttempt.amReq.setResourceName(ResourceRequest.ANY);
        appAttempt.amReq.setRelaxLocality(true);

        // SchedulerUtils.validateResourceRequests is not necessary because
        // AM resource has been checked when submission
        Allocation amContainerAllocation =
            appAttempt.scheduler.allocate(appAttempt.applicationAttemptId,
                Collections.singletonList(appAttempt.amReq),
                EMPTY_CONTAINER_RELEASE_LIST, null, null); //分配资源
        //无需要释放的容器,在我们这个情景中 scheduler 是 FifoScheduler
        if (amContainerAllocation != null
            && amContainerAllocation.getContainers() != null) {
            assert (amContainerAllocation.getContainers().size() == 0);
        }
        return RMAppAttemptState.SCHEDULED; //RMAppAttemptImpl 状态机的新状态
    } else {
        // save state and then go to LAUNCHED state
        appAttempt.storeAttempt();
        return RMAppAttemptState.LAUNCHED_UNMANAGED_SAVING;
    }
}
```

可见, `ScheduleTransition.transition()` 的返回值因 `subCtx.getUnmanagedAM()` 的结果而异。这个 `subCtx` 就是用户提交的 `ApplicationSubmissionContext`。如果 `subCtx.getUnmanagedAM()` 的结果为 `false`, 表示这不是一个 `UnmanagedAM`, 最后就会返回 `SCHEDULED`; 否则就会返回

LAUNCHED_UNMANAGED_SAVING。总之, `ScheduleTransition.transition()` 返回什么, 状态机的下一个状态就是什么。

所谓“不受管理的 AM(UnmanagedAM)”是指不由 RM 发起和管理, 而直接由使用者在 RM 节点上通过命令行发起的 AM。Hadoop 的代码中有个类, 叫 `UnmanagedAMLauncher`, 这个类是有 `main()` 函数的, 所以使用者可以直接通过命令行发起一个 JVM 来执行这个类, 这样就绕过了常规的作业提交和调度, 直接就进入(在某个 `NodeManager` 节点上)发起 `AppMaster` 的阶段。不过这样的 `AppMaster` 一经发起后仍得向 RM 登记, 因为所需的资源总归还得由 RM 分配。我们在这里只关心常规的情景和流程, 有兴趣的读者可以自己阅读和分析 `UnmanagedAMLauncher` 的代码。

但是, 为什么在这个地方会涉及 `UnmanagedAM` 呢? 这是因为, 对于正常提交的 App, 更准确地说是 `AppAttempt`, 下一步要做的也正是要指派在某个 `NodeManager` 节点上发起一个起着“课题组长”作用的 `AppMaster`, 这一点上二者的走向是共同的。虽然 `AppMaster` 并不直接执行具体 App 的程序, 它也是一个独立运行的任务, 是要启动一个独立的 JVM 加以执行的, 它也需要耗用一定的资源。所以, 要指派发起 `AppMaster`, 就得为其分配一个“容器(Container)”。这里所做的, 就是先填写好一个 `ResourceRequest`, 就是代码中的 `amReq`, 然后通过调度器的 `allocate()` 方法分配容器。注意, 这里要求分配的容器数量是 1, 那就是 `AppMaster` 本身所需的容器。至于 `AppMaster` 发起执行具体 `AppAttempt` 的也许是大量的任务(例如 `Mapper` 和 `Reducer`)时所需的(也许是大量的)容器, 那是以后的事, 得由 `AppMaster` 出面向调度器申请了。

这个 `transition()` 函数与我们在前面看到的那些有所不同。一方面, 是它返回一个状态, 不是 `SCHEDULED` 就是 `LAUNCHED_UNMANAGED_SAVING`, 因为这是“多弧跳变”, 具体跳变到什么状态要视情况而定。另一方面, `transition()` 函数通常会在末尾发出一个别的什么事件, 这个事件会被用来触发另一个过程, 往往回过头来又使这个状态机发生变化, 这就构成驱动状态机运转的动力, 但是这里却没有。这是为什么呢? 其实原因也很简单, 因为这个状态机之是否还可前行, 要看通过调度器分配资源能否成功。这里的 `scheduler.allocate()`, 对于我们来说就是 `FifoScheduler.allocate()`。下面我们就将看到, 在那里, 每拿到一个容器就会发出一个 `RMContainerEventType.ACQUIRED` 事件。这个事件最终会导致一个 `RMAppAttemptEventType.CONTAINER_ALLOCATED` 事件, 使这状态机跳出 `SCHEDULED` 状态, 经由 `ALLOCATED_SAVING` 而进入 `ALLOCATED` 状态。但是, 如果一时分配不到容器, 则这个 `RMAppAttemptImpl` 的容器就会暂时停滞在 `SCHEDULED` 状态。

前面看到, 当一个 App 被提交到 `RMAppManager` 手里时, `RMAppManager.submitApplication()` 通过 `createAndPopulateNewRMApp()` 创建了一个 `RMAppImpl` 对象, 此后又创建一个 `RMAppAttemptImpl` 对象作为运行这个 App 的一次尝试, 后者则在调度器中创建一个 `FiCaSchedulerApp` 对象, 作为其在调度器中的代表。

现在, 由于受 `ATTEMPT_ADDED` 事件的触发, 这个具体的 `RMAppAttemptImpl` 对象 `appAttempt` 要求调度器分配资源, 这样就进入了 `FifoScheduler.allocate()`。

```
[RMAppManager.submitApplication() => RMAppNewlySavingTransition.transition()
=> AddApplicationToSchedulerTransition.transition() => StartAppAttemptTransition.transition()]
```



```

=> AttemptStartedTransition.transition() => FifoScheduler.addApplicationAttempt()
=> RMAppAttemptImpl.ScheduleTransition.transition() > FifoScheduler.allocate()]

FifoScheduler.allocate(ApplicationAttemptId applicationAttemptId,
    List<ResourceRequest> ask, List<ContainerId> release,
    List<String> blacklistAdditions, List<String> blacklistRemovals)
> FiCaSchedulerApp application = getApplicationAttempt(applicationAttemptId)
    //代表着要求分配资源的 AppAttempt
> SchedulerUtils.normalizeRequests(ask, resourceCalculator, clusterResource,
    minimumAllocation, maximumAllocation) //资源要求的合理性检测和规格化
> releaseContainers(release, application) //释放该释放的容器
> if (!ask.isEmpty()) { //要求分配的资源集合非空
>+ application.updateResourceRequests(ask) // Update application requests
> }
> application.updateBlacklist(blacklistAdditions, blacklistRemovals) //修改黑名单
> ContainersAndNMTokensAllocation allocation =
    application.pullNewlyAllocatedContainersAndNMTokens()
    //把新分配的容器都收揽过来并为之创建 Token
>> for (Iterator<RMContainer> i = newlyAllocatedContainers.iterator(); i.hasNext();) {
    //对于 newlyAllocatedContainers 列表中的每一个容器
>>+ RMContainer rmContainer = i.next()
>>+ Container container = rmContainer.getContainer()
>>+ return this.container
>>+ container.setContainerToken(
    rmContext.getContainerTokenSecretManager().createContainerToken(
        container.getId(), container.getNodeId(), getUser(), container.getResource(),
        container.getPriority(), rmContainer.getCreationTime(), this.logAggregationContext))
>>+ NMToken nmToken = rmContext.getNMTokenSecretManager().createAndGetNMToken(
    getUser(), getApplicationAttemptId(), container)
>>+ if (nmToken != null) nmTokens.add(nmToken)
>>+ returnContainerList.add(container) //揽到了一个容器
>>+ i.remove() //从 newlyAllocatedContainers 列表中摘除
>>+ rmContainer.handle(new RMContainerEvent(
    rmContainer.getContainerId(), RMContainerEventType.ACQUIRED))
>> } //end for
    //已从该 App 的 newlyAllocatedContainers 中把容器都揽到了 returnContainerList 中
>> return new ContainersAndNMTokensAllocation(returnContainerList, nmTokens)
    //所返回的 ContainersAndNMTokensAllocation 对象成为前面的 allocation
> return new Allocation(allocation.getContainerList(), application.getHeadroom(),
    null, null, null, allocation.getNMTokenList())

```

这个函数有 5 个参数,第一个是 `applicationAttemptId`,其意义自明。此外就是 4 个 `List`,其中 `ask` 是资源请求列表;`release` 是资源释放列表;`blacklistAdditions` 和 `blacklistRemovals` 则分别是需要加入黑名单或从中移除的资源列表。如果一个节点或机架在一个 App 的黑名单中,调度器就不会在这个节点或机架上为这个 App 分配资源。这样,如果需要,就可以让一个 App 有意避开在某些节点或机架上运行。不过,我们在前面看到,从 `ScheduleTransition.transition()` 中调用 `allocate()` 函数时这两个参数都是 `null`。另外,调用时具体的要求是只要一个容器。

所以,这个函数既可用于分配资源,也可用于释放资源。所谓资源的分配和释放,实际上就是容器的分配与释放,因为具体的资源,例如存储空间和 `VCORE`,是包裹(记载)在容器里面的。

我们只关心容器的分配,故而直接看对于 `pullNewlyAllocatedContainersAndNMTokens()` 的调用。这个函数定义于 `SchedulerApplicationAttempt` 类内部,而 `FiCaSchedulerApp` 是对于 `SchedulerApplicationAttempt` 的扩充,所以这也是由 `FiCaSchedulerApp` 提供的操作方法。另外,`SchedulerApplicationAttempt` 即 `FiCaSchedulerApp` 内部还有个 `RMContainer` 的列表 `newlyAllocatedContainers`,意为新分配的容器。在这个列表中,如果有的话,是根据 `FiCaSchedulerApp` 的要求为其分配的容器(可以有多个),这些容器可以来自不同的节点(更确切地说是可以由不同的节点提供)。我们暂且不问这些容器的来历,只要知道这个列表中可能有分配好的容器就行了。而所谓 `allocate()`,其实只是从这个列表中收揽已经分配的容器(`RMContainer` 对象),并为这些容器办理使用证件(`NMTOKEN` 对象),然后一并打包成一个 `Allocation` 对象。其中收揽容器和办理证件的操作就是由 `pullNewlyAllocatedContainersAndNMTokens()` 完成的。

特别地,每当从 `newlyAllocatedContainers` 列表中收揽到一个容器,就向这容器(更确切地说是该容器的状态机)发出一个 `RMContainerEventType.ACQUIRED` 事件。

既然这些容器已经存在于 `newlyAllocatedContainers` 这个 `List` 中,当然是已经被创建的了。每个 `RMContainer` 对象都有自己的状态机,因为容器有它的生存周期,有状态变迁。

但是,也有可能在 `newlyAllocatedContainers` 中还没有所需的容器,那样就不会有 `ACQUIRED` 事件发出,`RMAppAttemptImpl` 对象的状态机就会停滞在 `SCHEDULED` 状态。可想而知,系统中一定有一种机制,使得当有容器可供分配时再走一遍上述的过程。后面我们将会看到这一点。

现在得说一下这些容器的来源了。

6.2 NM 节点的心跳和容器周转

在 Hadoop 集群中,每个 NM 节点每过一会儿就会向 RM 结点发送一次“心跳”,即 `Heartbeat` 信息。这个 `Heartbeat` 信息不仅仅表示“我还活着”,还搭载着许多关于本节点的状态信息,所以其实就是定期向 RM 节点发出的情况汇报。在 `NodeManager` 内部,或者说在运行着 `NodeManager` 的 JVM 上,有个属于 `NodeStatusUpdaterImpl` 的线程,这个线程周期地收集本节点的状态信息,并向 RM 节点发送心跳。

```
NodeStatusUpdaterImpl.statusUpdaterRunnable.run()  
> while (!isStopped) { //只要没有停止命令,就一直循环
```

```

>+ NodeStatus nodeStatus = getNodeStatus(lastHeartBeatID) //收集本节点的状态信息
>+ request = NodeHeartbeatRequest.newInstance(nodeStatus, ...) //准备好一个心跳报告
>+ response = resourceTracker.nodeHeartbeat(request)
      == ResourceTracker.nodeHeartbeat() //向 RM 节点发送心跳报告,并接收回应
>+ // 然后根据 RM 节点的回应进行处理,执行来自 RM 节点的命令
>+ nextHeartBeatInterval = response.getNextHeartBeatInterval()
>+ updateMasterKeys(response)
>+ if (response.getNodeAction() == NodeAction.SHUTDOWN) {
>+ e = new NodeManagerEvent(NodeManagerEventType.SHUTDOWN)
>+ dispatcher.getEventHandler().handle(e)
>+ }
>+ if (response.getNodeAction() == NodeAction.RESYNC) {
>+ e = new NodeManagerEvent(NodeManagerEventType.RESYNC)
>+ dispatcher.getEventHandler().handle(e)
>+ }
>+ // 根据命令清除不再需要运行的容器和应用
>+ removeOrTrackCompletedContainersFromContext(
      response.getContainersToBeRemovedFromNM())
>+ lastHeartBeatID = response.getResponseId()
>+ List<ContainerId> containersToCleanup = response.getContainersToCleanup()
>+ e = new CMgrCompletedContainersEvent(containersToCleanup,
      CMgrCompletedContainersEvent.Reason.BY_RESOURCEMANAGER)
>+ List<ApplicationId> appsToCleanup = response.getApplicationsToCleanup()
>+ trackAppsForKeepAlive(appsToCleanup)
>+ e = new CMgrCompletedAppsEvent(appsToCleanup,
      CMgrCompletedAppsEvent.Reason.BY_RESOURCEMANAGER)
>+ dispatcher.getEventHandler().handle(e)
>+ heartbeatMonitor.wait(nextHeartBeatInterval) //睡眠一段时间后再进入下一轮循环
> } //end while

```

RM 节点上与 NM 节点对接的是 ResourceTrackerService, 这个模块通过接收心跳跟踪各个 NM 节点的资源使用。

当然, NM 节点上的那个线程不可能直接调用 RM 节点上的 nodeHeartbeat(), 中间需要用到 Hadoop 的 RPC 机制。上面的 ResourceTracker 就是通向 RM 的 proxy, 所以对 ResourceTracker.nodeHeartbeat() 的调用实质上就是对 RM 的 RPC 调用, 这使 RM 节点上的 ResourceTrackerService.nodeHeartbeat() 受到调用。下面是 RM 节点上接收到来自一个 NM 节点的心跳信息时的操作摘要。

```

ResourceTrackerService.nodeHeartbeat(NodeHeartbeatRequest request)
> remoteNodeStatus = request.getNodeStatus() //从心跳报告中获取所搭载的状态信息
> nodeId = remoteNodeStatus.getNodeId() //必须知道是从哪个节点来的

```

```

> rmNode = this.rmContext.getRMNodes().get(nodeId) //找到 RM 节点上相应的对象 rmNode
> this.nmLivelinessMonitor.receivedPing(nodeId) //告知 LivelinessMonitor,这个节点还活着
> //此处略去了对节点和心跳报告的合法性检查
> nodeHeartBeatResponse = YarnServerBuilderUtils.newNodeHeartbeatResponse(...) //准备回应
> rmNode.updateNodeHeartbeatResponseForCleanup(nodeHeartBeatResponse) //需清除的容器
> populateKeys(request, nodeHeartBeatResponse) //设置其他回应内容
> e = new RMNodeStatusEvent(nodeId, remoteNodeStatus.getNodeHealthStatus(), ...)
>> super(nodeId, RMNodeEventType.STATUS_UPDATE)
> this.rmContext.getDispatcher().getEventHandler().handle(e)
//以此事件驱动具体 RMNodeImpl 的状态机
>> StatusUpdateWhenHealthyTransition.transition(rmNode, event)
//这是伴随着 RMNodeImpl 状态机跳变的操作
> return nodeHeartBeatResponse //RPC 机制会发送这个回应

```

RM 节点上对于本集群内所有已注册的 NM 节点都维持着一个 RMNodeImpl 对象,每当接收到来自一个节点的心跳报告,就根据报告的内容更新相应 RMNodeImpl 对象的内容,并向其状态机发送一个 STATUS_UPDATE 事件,然后通过 Hadoop 的 RPC 机制发回一个对心跳报告的回应。跟心跳报告相似,在回应中也可搭载一些信息和命令,主要是关于清除不再需要运行的容器和应用。读者可以与前面 NM 节点上那个线程的摘要比照参考。

具体 RMNodeImpl 对象的状态机对于 STATUS_UPDATE 事件的跳变规则是这样的:

```

addTransition(NodeState.RUNNING,
    EnumSet.of(NodeState.RUNNING, NodeState.UNHEALTHY),
    RMNodeEventType.STATUS_UPDATE,
    new StatusUpdateWhenHealthyTransition())

```

就是说,这个状态机的状态可能继续停留在 RUNNING,也可能变成 UNHEALTHY,视 StatusUpdateWhenHealthyTransition.transition() 的返回值而定,实际上取决于事件报告的内容。StatusUpdateWhenHealthyTransition.transition() 是定义于 RMNodeImpl 内部的一个类:

```

[ResourceTrackerService.nodeHeartbeat() => StatusUpdateWhenHealthyTransition.transition()]

```

```

StatusUpdateWhenHealthyTransition.transition(RMNodeImpl rmNode,
    RMNodeEvent event)
> statusEvent = (RMNodeStatusEvent) event
> rmNode.latestNodeHeartBeatResponse = statusEvent.getLatestResponse()
> remoteNodeHealthStatus = statusEvent.getNodeHealthStatus()
> rmNode.setHealthReport(remoteNodeHealthStatus.getHealthReport())
> if (!remoteNodeHealthStatus.getIsNodeHealthy()) { //如果这个节点已经不健康了
>+ rmNode.nodeUpdateQueue.clear()
>+ e = new NodeRemovedSchedulerEvent(rmNode)

```

```

>+> super(SchedulerEventType.NODE_REMOVED)
>+ rmNode.context.getDispatcher().getEventHandler().handle(e)
// 告知 NodesListManager, 这个节点不再可用
>+ e = new NodesListManagerEvent(
NodesListManagerEventType.NODE_UNUSABLE, rmNode)
>+ rmNode.context.getDispatcher().getEventHandler().handle(e)
>+ return NodeState.UNHEALTHY //RMNodeImpl 的状态机转入“不健康”状态
> }
> //这个节点依然健康, 可以有进一步的操作
> List<ContainerStatus> newlyLaunchedContainers = new ArrayList<ContainerStatus>()
> List<ContainerStatus> completedContainers = new ArrayList<ContainerStatus>()
> for (ContainerStatus remoteContainer : statusEvent.getContainers()) {
//扫描本事件所涉及的每个容器
>+ containerId = remoteContainer.getContainerId()
>+ if (remoteContainer.getState() == ContainerState.RUNNING) { //这个容器已经在运行中
>++ if (!rmNode.launchedContainers.contains(containerId)) { //但不在该节点的已发起名单中
>+++ rmNode.launchedContainers.add(containerId) //加入该节点的已发起名单
>+++ newlyLaunchedContainers.add(remoteContainer) //这是一个新发起的容器
>++ }
>+ } else { // A finished container, 这个容器的运行已经结束
>++ rmNode.launchedContainers.remove(containerId) //从该节点的已发起名单中去掉
>++ completedContainers.add(remoteContainer) //转入已结束容器名单
>+ }
> } //end for
> if(newlyLaunchedContainers.size() != 0 || completedContainers.size() != 0) {
>+ //有容器改变了状态, 把有关这些容器变化的信息收集在该节点的 nodeUpdateQueue 中
>+ u = new UpdatedContainerInfo(newlyLaunchedContainers, completedContainers)
>+ rmNode.nodeUpdateQueue.add(u) //有状态变化的容器队列, 后面要用到
> }
> if(rmNode.nextHeartBeat) { //至少有一个容器的状态发生了变化
>+ rmNode.nextHeartBeat = false //以保证只发生一次
>+ e = new NodeUpdateSchedulerEvent(rmNode)
>+> super(SchedulerEventType.NODE_UPDATE) //通知调度器有节点发生了状态变化
>+ rmNode.context.getDispatcher().getEventHandler().handle(e)
>+> ResourceSchedulerWrapper.handle(SchedulerEventType.NODE_UPDATE)
>+>> scheduler.handle() == FifoScheduler.handle(SchedulerEventType.NODE_UPDATE)
>+>>> nodeUpdatedEvent = (NodeUpdateSchedulerEvent)event
>+>>> nodeUpdate(nodeUpdatedEvent.getRMNode()) //FifoScheduler.nodeUpdate(), 见后
> }
> return NodeState.RUNNING //这个 RMNodeImpl 的状态机仍停留在“运行”状态

```

摘要中已加了注释,就不用再解释了。这段程序最后对调度器发出 `NODE_UPDATE` 事件,不过调度器并不使用状态机,而只是在其 `handle()` 函数中通过 `switch` 语句分情形处理。我们在前面已经看过 `FifoScheduler.handle()` 的摘要,但那时我们并不关心 `NODE_UPDATE`,所以现在要另外为此专门做个摘要:

```
[ResourceTrackerService.nodeHeartbeat() => StatusUpdateWhenHealthyTransition.transition()
> FifoScheduler.handle()]
```

```
FifoScheduler.handle(SchedulerEvent event)
> switch(event.getType()) {
> case NODE_UPDATE:
>+ NodeUpdateSchedulerEvent nodeUpdatedEvent = (NodeUpdateSchedulerEvent)event
>+ nodeUpdate(nodeUpdatedEvent.getRMNode()) // FifoScheduler.nodeUpdate()
> }
```

就是说,只要来自某个 NM 节点的心跳报告表明该节点上的容器使用有了变化,调度器就会执行其 `nodeUpdate()` 函数。就 `FifoScheduler` 而言,这就是 `FifoScheduler.nodeUpdate()`。下面是它的摘要:

```
[ResourceTrackerService.nodeHeartbeat() => StatusUpdateWhenHealthyTransition.transition()
=> FifoScheduler.nodeUpdate()]
```

```
FifoScheduler.nodeUpdate(RMNode rmNode)
> FiCaSchedulerNode node = getNode(rmNode.getNodeID())
> List<UpdatedContainerInfo> containerInfoList = rmNode.pullContainerUpdates()
//来自前面的 rmNode.nodeUpdateQueue
> List<ContainerStatus> newlyLaunchedContainers
> List<ContainerStatus> completedContainers
> //containerInfoList 来自该节点的 nodeUpdateQueue,包括新发起和已结束这两种容器
> for(UpdatedContainerInfo containerInfo: containerInfoList) {
>+ newlyLaunchedContainers.addAll(containerInfo.getNewlyLaunchedContainers()) //新发起
>+ completedContainers.addAll(containerInfo.getCompletedContainers()) //已结束
> }
> //对这两种容器分别加以处理
> for (ContainerStatus launchedContainer : newlyLaunchedContainers) {
//对于每个新发起的容器
>+ containerLaunchedOnNode(launchedContainer.getContainerId(), node)
>+> RMContainer rmContainer = getRMContainer(containerId) //根据 ID 找到这个容器对象
>+> e = new RMContainerEvent(containerId, RMContainerEventType.LAUNCHED)
>+> rmContainer.handle(e) //使该容器的状态机受 LAUNCHED 事件的触发
> }
> for (ContainerStatus completedContainer : completedContainers) {
```



```

//对于每个已结束的容器(会有资源释放)
>+ containerId = completedContainer.getContainerId()
>+ LOG.debug("Container FINISHED: " + containerId)
>+ completedContainer(getRMContainer(containerId),
                        completedContainer, RMContainerEventType.FINISHED)
>+> Container container = rmContainer.getContainer()
>+> FiCaSchedulerApp application = getCurrentAttemptForContainer(container.getId())
>+> appId = container.getId().getApplicationAttemptId().getApplicationId()
>+> FiCaSchedulerNode node = getNode(container.getNodeId())
                        //Get the node on which the container was allocated
>+> application.containerCompleted(rmContainer, containerStatus, event) // FINISHED
                        == FiCaSchedulerApp.containerCompleted()
>+>> liveContainers.remove(rmContainer.getContainerId())
>+>> newlyAllocatedContainers.remove(rmContainer)
                        //Remove from the list of newly allocated containers if found
>+>> Container container = rmContainer.getContainer()
>+>> containerId = container.getId()
>+>> e = new RMContainerFinishedEvent(containerId, containerStatus, event)
                        //这个事件就是 RMContainerEventType.FINISHED
>+>> rmContainer.handle(e) //Inform the container
>+>> containersToPreempt.remove(rmContainer.getContainerId())
>+>> Resource containerResource = rmContainer.getContainer().getResource()
                        //该容器所占资源,即 VCores 和 Memory
>+>> Resources.subtractFrom(currentConsumption, containerResource)
                        //从 currentConsumption 中减去该容器所占用的资源
>+> node.releaseContainer(container)
>+> Resources.subtractFrom(usedResource, container.getResource())
                        //从 usedResource 中减去该容器所占用的资源
> } //end for
> if (Resources.greaterThanOrEqualTo(resourceCalculator,
                                clusterResource, node.getAvailableResource(), minimumAllocation)) {
                                //若可供分配的资源达到了门槛值 minimumAllocation;
>+ LOG.debug("Node heartbeat " + rmNode.getNodeID() +
            " available resource = " + node.getAvailableResource())
>+ assignContainers(node) //分配容器。这是我们关注的焦点
>+ LOG.debug("Node after allocation " + rmNode.getNodeID()
            + " resource = " + node.getAvailableResource())
> } //end if
> updateAvailableResourcesMetrics()

```

来自 NM 节点的心跳报告中搭载着该节点上容器变化的信息,即哪些容器结束了运行,

哪些容器又发起了运行。前面已经把这些信息收集在该节点的 `nodeUpdateQueue` 队列中,现在就来做进一步的处理。代码中的两个 `for` 循环就是对发生了变化的这两种容器的处理。

对于刚发起运行的容器,这里要做的是向代表着这个容器的 `RMContainerImpl` 发送一个 `LAUNCHED` 事件,使其状态机发生相应的跳变,并完成所需的伴随操作。

而对于刚结束运行的容器,则除了向其发送 `FINISHED` 事件之外还有一系列“账务”上的处理,将其绑定的资源释放出来。

经过这样的处理,如果这个节点上可供分配的资源达到了最低限度的要求,就可以通过 `assignContainers()` 分配容器了。

6.3 容器的分配

容器(container)这个词的本来意义是用来盛放资源的器皿和包裹,但是因此也就有了另一种意义,就是配套成组的资源,其实也就是几个 `VCORE` 和多大内存,把这些资源打包在一起就是一个容器,所以容器的分配就是成组资源的分配。不同 App 的不同任务,其所需要的资源也是不同的,所以每个 App 的不同任务所需的容器原则上都需要量身定制,而不能预先就批量生产好。显然,容器分配的核心在于资源,容器本身即 `Container` 对象只起着包装盒的作用。有没有容器可以分配,问题不在于有没有包装盒,而在于有没有足够的资源可以满足具体的需要而凑成一盒。

在 Hadoop 的源码中,`Container` 的定义确实有点让人困惑。

首先,在 `org.apache.hadoop.yarn.api.records` 这个 package 中定义了一个抽象类 `Container`,但是这更像是个界面定义,只是定义了一些方法,而并无关于数据成分的定义。它没有提供显式的构造方法,但提供了方法 `newInstance()`:

```
abstract class Container implements Comparable<Container> {}

] newInstance(ContainerId containerId, NodeId nodeId, String nodeHttpAddress,
               Resource resource, Priority priority, Token containerToken)
  > Container container = Records.newRecord(Container.class)
  > container.setId(containerId)           //记录块中有 containerId
  > container.setNodeId(nodeId)           //记录块中有 Container 所在节点的 nodeId
  > container.setNodeHttpAddress(nodeHttpAddress) //还有该节点的 Http 地址
  > container.setResource(resource)        //容器中所含的资源
  > container.setPriority(priority) //优先级, PRIORITY_MAP 高于 PRIORITY_REDUCE
  > container.setContainerToken(containerToken) //用于访问控制的 Token
  > return container
```

之所以如此,是因为它只是笼统地定义了一种(用于通信的)记录块,或者说报文的格式,只是说记录块中应该有些什么信息。至于记录块的确切格式、具体的编码方式,则还要看具体的协议。由于 Hadoop 采用 `ProtoBuf`,相应的具体类就是 `ContainerPBImpl`。

与此类似,还有 `Resource` 的定义也是这样:

```
abstract class Resource implements Comparable<Resource> {}
```

```

] newInstance(int memory, int vCores)
  > Resource resource = Records.newRecord(Resource.class)
  > resource.setMemory(memory)           // 内存空间大小
  > resource.setVirtualCores(vCores)     // 虚核 VCore 的个数
  > return resource

```

同样,相应的具体类是 ResourcePBIImpl。

当 RM 节点上的资源调度器应 AM 的要求分配容器和资源的时候,它只是创建一个记录块作为响应报文的一部分发回给 AM。就 RM 节点而言,它并不需要记住一个个具体的容器,甚至也不必有“容器”的概念,它只要知道在什么节点上还有多少资源就行了。

但是到了 NM 节点上,特别是在 AM 那里,就需要有具体容器的概念了。所以 Hadoop 的代码中定义了一个界面,也叫 Container:

```

interface Container extends EventHandler<ContainerEvent> {}
] ContainerId getContainerId()
] Resource getResource()
] String getUser()
] ContainerTokenIdentifier getContainerTokenIdentifier()
] ...

```

但这是 org.apache.hadoop.yarn.server.nodemanager.containermanager.container,这是在另一个 package 中。实现这个界面的类是 ContainerImpl,它的数据部分的摘要为:

```

class ContainerImpl implements Container {}
] ContainerLaunchContext launchContext
] ContainerId containerId
] Resource resource           //org.apache.hadoop.yarn.api.records.Resource
] String user
] Map<LocalResourceRequest,List<String>> pendingResources
] Map<Path,List<String>> localizedResources           //已经本地化的资源
] List<LocalResourceRequest> publicRsrcs             //公共资源
] List<LocalResourceRequest> privateRsrcs            //私有资源
] List<LocalResourceRequest> appRsrcs                //App 资源
] StateMachine<ContainerState, ContainerEventType, ContainerEvent> stateMachine

```

显然这个类的对象中含有更多的信息,这是由 AM 综合进去的。注意,作为具体容器的 ContainerImpl 对象总是针对具体节点的。

所以,我们在源码中看到引用 Container 这个类型时,需要看一下文件头部的那些 import 语句,看导入的是哪一个 package 中的 Container。

就此刻而言,我们要分配的容器是建立具体 App 的 AppMaster 所需,那相当于项目组长的作用所需的开销,自然也应该算是具体 App 的一项任务,但是与 App 本身正式的任务,如 Mapper 和 Reducer,却并无直接联系,所用资源也与那些任务无关。

针对具体节点的容器分配是由具体调度器的 `assignContainers()` 完成的。

在实际存在的集群中,节点是有地域之分的,因为至少每台机器都安装在某个机架上,同一个机架上有好几台机器,它们之间的通信效率可能高一些。另外,更重要的是,一般而言应该把一个应用的容器分配在哪里,让它在哪里运行,要看它的数据在哪里,这就是“计算跟着数据走”的原则。所以,App 的资源需求一般都是有地域要求的,Hadoop 为此定义了三种地域 (Locality) 类型,即 `NODE_LOCAL`、`RACK_LOCAL` 和 `OFF_SWITCH`。App 在其资源要求中可以给定节点名 (NodeName) 或机架名 (RackName), 或使用表示“任意”的常数 `ResourceRequest.ANY`, 分别表示指定节点、指定机架或任意。另外,App 对于资源的要求也可以有优先级别之分,不过那只是同一个 App 的不同任务之间的优先级,而不是不同 App 之间的优先级。例如 `PRIORITY_MAP` 的级别就比 `PRIORITY_REDUCE` 高,因为 Reducer 依赖于 Mapper,Mapper 不完成 Reducer 就无事可做。

那么 RM 节点上的资源调度器怎么知道应该把具体 App 的具体任务分配到什么节点上呢? 属于同一 App 的不同 Mapper, 它们的输入数据很可能就在不同的节点上, RM 怎样才能让这些任务跟着数据走呢? 其实这并不需要 RM 操心,这是 AM 的事,AM 在资源请求中就指明需要哪些节点上的容器, RM 只要尽量满足要求就是了。为此我们不妨看一下 `ContainerRequest` 这个类的定义摘要,这是定义在抽象类 `RMContainerRequestor` 的内部:

```
class RMContainerRequestor.ContainerRequest {
    ] TaskAttemptId attemptID
    ] Resource capability    //要求什么资源
    ] String[] hosts        //要求在哪一些节点机上
    ] String[] racks        //要求在哪一些机架上
```

当然也可以不指定节点和机架,那就把 `hosts` 和 `racks` 设置成 `null`。

了解这些情况以后,我们就可以看 `FifoScheduler.assignContainers()` 的代码摘要了。记住,这是针对一个特定的节点为已经受理的那些 App 分配容器。其中,有些 App 可能本来就要求在这个节点上分配容器(类型为 `NODE_LOCAL` 且节点地址相符);有些 App 只是说也可以在这个节点上分配容器(类型为 `RACK_LOCAL` 且机架名相符,或类型为 `OFF_SWITCH`);有些则不愿意在这个节点上分配容器(类型为 `NODE_LOCAL` 但节点地址不符,或类型为 `RACK_LOCAL` 但机架名不符)。然而也要看到,不管是什么 App,一般而言其 AppMaster 可以放在任何节点上,因为作为项目组长的 AppMaster 并不亲自处理数据。

```
[ResourceTrackerService.nodeHeartbeat() => StatusUpdateWhenHealthyTransition.transition()
=> FifoScheduler.nodeUpdate() > FifoScheduler.assignContainers()]

FifoScheduler.assignContainers(FiCaSchedulerNode node)
> for (Map.Entry<ApplicationId, SchedulerApplication<FiCaSchedulerApp>>
    e : applications.entrySet()) {
    //对于调度器中的每一个 App,试图满足其对于容器的要求
    >+ FiCaSchedulerApp application = e.getValue().getCurrentAppAttempt()
    //获取该 App 的当前 AppAttempt 对象
```

```

>+ if (SchedulerAppUtils.isBlacklisted(application, node, LOG)) continue
    //如果这个 App 不适合放在这个节点上,已将本节点列入其黑名单,就跳过
>+ for (Priority priority : application.getPriorities()) {
    //对于同一 App 不同优先级别的容器要求
>++ maxContainers = getMaxAllocatableContainers(application,
    priority, node, NodeType.OFF_SWITCH)
>++ if (maxContainers > 0) {
>+++ int assignedContainers = assignContainersOnNode(node, application, priority)
>+++> nodeLocalContainers = assignNodeLocalContainers(node, application, priority)
    //先尝试满足其指定需要在这个节点上的容器要求
>+++>> request = application.getResourceRequest(priority, node.getNodeName())
    //获取该 App 对于这个节点的资源要求
>+++>> if (request == null) return 0 //App 没有要求这个节点上的资源,就返回
>+++>> rackRequest = application.getResourceRequest(priority,
    node.getRMNode().getRackName())
    //获取该 App 对于这个机架的资源要求
>+++>> if (rackRequest == null || rackRequest.getNumContainers() <= 0) return 0
    //App 没有要求这个节点所在机架上的资源,就先不分配
>+++>> assignableContainers = Math.min(getMaxAllocatableContainers(application,
    priority, node, NodeType.NODE_LOCAL),
    request.getNumContainers())
>+++>> assignedContainers = assignContainer(node, application,
    priority, assignableContainers, request, NodeType.NODE_LOCAL)
>+++>> return assignedContainers //这是 assignNodeLocalContainers()返回的结果
>+++> rackLocalContainers = assignRackLocalContainers(node, application, priority)
    //再尝试满足可以在同一机架上的容器要求
>+++>> request = application.getResourceRequest(priority,
    node.getRMNode().getRackName())
>+++>> if (request == null) return 0 //App 没有在该机架上的容器要求
>+++>> offSwitchRequest = application.getResourceRequest(priority, ResourceRequest.ANY)
>+++>> if (offSwitchRequest.getNumContainers() <= 0) return 0
    //如果 App 仅在这一个机架上有容器要求,就先不分配
>+++>> assignableContainers = Math.min(getMaxAllocatableContainers(application,
    priority, node, NodeType.RACK_LOCAL),
    request.getNumContainers())
>+++>> assignedContainers = assignContainer(node, application,
    priority, assignableContainers, request, NodeType.RACK_LOCAL)
>+++>> return assignedContainers //这是 assignRackLocalContainers()返回的结果
>+++> offSwitchContainers = assignOffSwitchContainers(node, application, priority)
    //最后试图满足其余的容器要求

```

```

>+++>> request = application.getResourceRequest(priority, ResourceRequest.ANY)
>+++>> if (request == null) return 0      //没有进一步的容器要求就返回
>+++>> assignedContainers = assignContainer(node, application,
                                priority, request.getNumContainers(), request, NodeType.OFF_SWITCH)
>+++>> return assignedContainers //这是 assignOffSwitchContainers()返回的结果
>+++> return (nodeLocalContainers + rackLocalContainers + offSwitchContainers)
                                //这是 assignContainersOnNode()返回的结果
>+++ //已从 assignContainersOnNode()返回,三者之和为 assignedContainers
>+++ if (assignedContainers == 0) break
>++ } //end if (maxContainers > 0)
>+ } //end for (Priority priority : application.getPriorities())
                                //已完成对同一 App 所有优先级别资源要求的扫描
>+ if (Resources.lessThan(resourceCalculator, clusterResource,
                                node.getAvailableResource(), minimumAllocation)) break //资源不足就结束循环
> } //end for (Map.Entry<ApplicationId, SchedulerApplication<FiCaSchedulerApp>> ...)
                                //已完成对所有已受理 App 的扫描
> for (SchedulerApplication<FiCaSchedulerApp> application : applications.values()) {
>+ FiCaSchedulerApp attempt = (FiCaSchedulerApp) application.getCurrentAppAttempt()
>+ if (attempt == null) continue
>+ updateAppHeadRoom(attempt) //修改各个 FiCaSchedulerApp 的可用资源上限
>+> schedulerAttempt.setHeadroom(Resources.subtract(clusterResource, usedResource))
> } //end for

```

参数 node 是个 FiCaSchedulerNode 对象,代表着集群中从调度器角度看到的一个节点。作为调度器,它所关心的就是具体节点上资源使用的情况。

这个函数的核心是 assignContainersOnNode(),它是在两层 for 循环中得到调用的,第一层是对所有 App 的循环,第二层是对同一 App 的不同优先级的资源需求的循环。调用 assignContainersOnNode()的目的是在给定的节点上分配(生成)一个包含着一定资源(虚拟处理器核和内存)的容器,用来在这个节点上运行构成该 App 的某个任务(Task),这可以是一个 Mapper 或 Reducer,也可以是任何可以在宿主操作系统上作为进程运行的可执行程序。至于具体是哪个节点,一般就是向 RM 发来心跳报告表明发生了变化的那个节点。

从代码中可见,调用 assignContainersOnNode()的参数有三个,即 node、application 和 priority。其中,node 是目标节点在调度器中的代表,对于 FifoScheduler 就是一个 FiCaSchedulerNode 对象,它记载着这个节点上的资源使用情况,这是供给方。而 application 则是一个 App 在调度器中的代表,对于 FifoScheduler 就是一个 FiCaSchedulerApp 对象,记载着 App 的资源要求,这是需求方。而调度器则在双方之间进行协调。

正因为具体 App 的资源需求是有地域性的,assignContainersOnNode()分三步来尝试分配资源。第一步是通过 assignNodeLocalContainers()试图满足 App 指定在本节点上的需求(如果有的话)。第二步是通过 assignRackLocalContainers()试图满足 App 指定在本机架上的需求。最后才是通过 assignOffSwitchContainers()满足 App 允许灵活加以分配的需求。不

过,这三者最终都是通过更底层一些的 `assignContainer()` 加以实际分配。

注意,无论是 `assignNodeLocalContainers()` 还是 `assignRackLocalContainers()`,或者是 `assignOffSwitchContainers()`,都独立地通过 `getResourceRequest()` 获取具体 App 的容器要求,这个信息最终来自于相应的 `AppSchedulingInfo`。如果实际分配了容器,就会在 `assignContainer()` 里面对 `AppSchedulingInfo` 中的容器要求做出调整。所以,如果一些容器的要求在 `assignNodeLocalContainers()` 中得到了满足,就不会再出现在 `assignRackLocalContainers()` 所看到的容器要求中。这样,能在目标节点上分配(NodeLocal)的就不再跑到别的节点上,能在目标机架上分配(RackLocal)的就不再跑到别的机架上,实在不行才分配到不在同一机架(OffSwitch)的节点上。不管是哪一种情况,最终都是通过 `assignContainer()` 完成该种情况下的容器分配。

在上述的分配过程中数次调用了函数 `getMaxAllocatableContainers()`,这个函数计算在参数指定的范围(Node,Rack,或 OffSwitch)中最多能为本 App 分配多少个容器:

```
[ResourceTrackerService.nodeHeartbeat() => StatusUpdateWhenHealthyTransition.transition()
=> FifoScheduler.nodeUpdate() > assignContainers() > getMaxAllocatableContainers()]
```

```
getMaxAllocatableContainers(FiCaSchedulerApp application,
                             Priority priority, FiCaSchedulerNode node, NodeType type)
{
    > maxContainers = 0
    > ResourceRequest offSwitchRequest = //获得该 App 总的资源需求,不考虑地域
        application.getResourceRequest(priority, ResourceRequest.ANY)
    > if (offSwitchRequest != null) {
    > + maxContainers = offSwitchRequest.getNumContainers() //该 App 所需总的容器数量
    > }
    > if (type == NodeType.OFF_SWITCH) return maxContainers
    > if (type == NodeType.RACK_LOCAL) {
    > + ResourceRequest rackLocalRequest = //获得该 App 可以分配在此机架上的资源需求
        application.getResourceRequest(priority, node.getRMNode().getRackName())
        //node.getRMNode().getRackName()返回节点所在机架名称
    > + if (rackLocalRequest == null) return maxContainers
    > + maxContainers = Math.min(maxContainers, rackLocalRequest.getNumContainers())
        //该 App 可以分配在此机架上的容器数量
    > }
    > if (type == NodeType.NODE_LOCAL) {
    > + ResourceRequest nodeLocalRequest = //获得该 App 在此节点上的资源需求
        application.getResourceRequest(priority, node.getRMNode().getNodeAddress())
        //node.getRMNode().getNodeAddress()返回节点的地址
    > + if (nodeLocalRequest != null)
        maxContainers = Math.min(maxContainers, nodeLocalRequest.getNumContainers())
        //该 App 所需在此节点上的容器数量
    > }
```

```
> }
> return maxContainers
```

这里第一步先求得该 App 所需的总的容器数量,不考虑地域,以此作为 maxContainers 的初值。然后根据具体的类型加以修正,如果该 App 有些资源允许被分配落实在本节点所在的机架上,则将 maxContainers 修正成可以分配在这个机架上的容器数量。进一步,如果该 App 有指定要分配落实在本节点上的资源,则再将 maxContainers 修正成可以分配在这个节点上的容器数量。

最后,不管是 NodeLocal、RackLocal,还是 OffSwitch,最终都是通过 assignContainer() 完成本节点针对某种条件下的容器分配。

从函数名 assignContainers() 和 assignContainer() 看,似乎后者只是单个容器的分配,其实不然,这个函数实现的是一个节点为同一个 App 分配落实的同一优先级的所有容器。

```
[ResourceTrackerService.nodeHeartbeat() => StatusUpdateWhenHealthyTransition.transition()
=> FifoScheduler.nodeUpdate() > assignContainers() > assignContainersOnNode() >
assignNodeLocalContainers() > assignContainer()]

FifoScheduler.assignContainer(FiCaSchedulerNode node, FiCaSchedulerApp application,
    Priority priority, int assignableContainers, ResourceRequest request, NodeType type)
> Resource capability = request.getCapability() //这是具体 App 所要求的资源
> availableContainers = node.getAvailableResource().getMemory() / capability.getMemory()
    //从内存资源角度计算该节点上能容纳几份这样的任务
> assignedContainers = Math.min(assignableContainers, availableContainers)
    //需求和可能,取其小者
> for (int i = 0; i < assignedContainers; ++i) { //逐一分配这些容器
>+ nodeId = node.getRMNode().getNodeID()
>+ containerId = BuilderUtils.newContainerId(application.getApplicationAttemptId(),
    application.getNewContainerId()) //生成一个 ID
>+ Container container = BuilderUtils.newContainer(containerId, nodeId,
    node.getRMNode().getHttpAddress(), capability, priority, null)
>+> Container container = recordFactory.newRecordInstance(Container.class)
>+> container.setId(containerId) == ContainerPBImpl.setId(containerId) //余均类推
>+> container.setNodeId(nodeId)
>+> container.setNodeHttpAddress(nodeHttpAddress)
>+> container.setResource(resource)
>+> container.setPriority(priority)
>+> container.setContainerToken(containerToken)
>+> return container //创建一个容器,即 ContainerPBImpl 对象
>+ RMContainer rmContainer = application.allocate(type, node, priority, request, container)
    == FiCaSchedulerApp.allocate(type, node, priority, request, container)
    //为该容器创建 RMContainerImpl 对象
```

```

>+> if (getTotalRequiredResources(priority) <= 0) return null//参数不合理,不予分配
>+> rmContainer = new RMContainerImpl(container, this.getApplicationAttemptId(),
                                node.getNodeID(), appSchedulingInfo.getUser(), this.rmContext)
                                //创建一个 RMContainerImpl 对象,作为该容器在 RM 节点上的代表
>+>> this.stateMachine = stateMachineFactory.make(this) //构建该容器的状态机
>+>> this.containerId = container.getId()
>+>> this.nodeId = nodeId
>+>> this.container = container
>+>> this.appAttemptId = appAttemptId
>+>> this.eventHandler = rmContext.getDispatcher().getEventHandler()
>+>> this.isAMContainer = false
>+>> ...
>+> newlyAllocatedContainers.add(rmContainer) //所分配的容器都汇总在这里
>+> liveContainers.put(container.getId(), rmContainer)
>+> resourceRequestList = appSchedulingInfo.allocate(type, node, priority, request, container)
                                //已经分配了一个容器,对资源请求做相应调整
>+> Resources.addTo(currentConsumption, container.getResource()) //调整资源使用统计
>+> ((RMContainerImpl)rmContainer).setResourceRequests(resourceRequestList)
>+> e = new RMContainerEvent(container.getId(), RMContainerEventType.START)
>+> rmContainer.handle(e) //处理 RMContainerEventType.START 事件
>+> return rmContainer //从 FiCaSchedulerApp.Allocate()返回,分配了一个 RMContainer
>+ node.allocateContainer(rmContainer) // Inform the node
    //调整该节点上的容器和资源数量,并将此容器放入该节点的 launchedContainers 集合
>+ increaseUsedResources(rmContainer) //用于统计目的
> } //end for,完成了一个容器的分配,继续循环
> return assignedContainers //返回分配的容器数量

```

参数 node 和 application 的意义自明,参数 priority 表示为哪一个优先级的任务分配资源,assignableContainers 表示当前对于容器个数的总需求,request 是总的资源需求。至于 type,则是节点的地域类型,可以是 NODE_LOCAL、RACK_LOCAL 或 OFF_SWITCH。

每当一个节点的资源使用发生变化从而变得可以提供某些资源时,只要有需求存在,系统就会根据需要将可以提供的资源都分配出去,其结果是调度器的 assignContainer() 受到调用,至少为某些 App 产生了一些容器。容器在 RM 节点上的代表是 RMContainerImpl,每个 RMContainerImpl 对象都代表着具体的 App(更准确地说具体的 AppAttempt)对于具体节点上特定资源的使用权。所产生的容器暂时都在该 App,实际上是其 FiCaSchedulerApp 对象内部的新lyAllocatedContainers 集合中。

注意,这里所说的“节点”并非指物理上的节点,而是指 RM 节点上的调度器中代表着具体节点的 FiCaSchedulerNode 对象。而 App,则是指在调度器中代表着具体 App 的 FiCaSchedulerApp 对象。

而所谓容器,情况就复杂了。从代码中可以看出,在创建容器的时候,首先创建的是一个

Container 类对象。但是 Container 类只是个抽象类,所以实际创建的只能是落实并扩充而成的 ContainerPBImp 类对象。这 ContainerPBImp 类对象就是将来要跨节点传输的“容器”,实际上是个记录块。所以,这里的 Container 类对象既不是通过 new 操作创建,也不是通过 newInstance() 创建,而是通过 recordFactory.newRecordInstance() 创建的。另外,如果看一下这两个类的 java 文件前面的 package 语句,可以发现它们都属于 org.apache.hadoop.yarn.api.records 这个 package。另外,在创建了这个容器以后还要调用其所属 FiCaSchedulerApp 对象的 allocate() 函数,以创建该容器在 RM 节点上的代表,即 RMContainerImpl 对象。正是这 RMContainerImpl 对象的状态机,反映着这个容器在目标节点上的状况和生存周期。

RMContainerImpl 对象初创时其状态机处于初始状态 NEW,然后 assignContainer() 在完成了一些准备后就向这状态机发出 RMContainerEventType.START 事件。而这状态机的相应跳变规则为:

```
addTransition(RMContainerState.NEW, RMContainerState.ALLOCATED,
              RMContainerEventType.START, new ContainerStartedTransition())
```

所以,这个容器的状态机在执行 ContainerStartedTransition.transition() 以后就会将状态变成 ALLOCATED。

```
[FifoScheduler.assignContainer() => RMContainerImpl.ContainerStartedTransition.transition()]
```

```
ContainerStartedTransition.transition(RMContainerImpl container, RMContainerEvent event)
> e = new RMAppAttemptContainerAllocatedEvent(container.appAttemptId)
>> super(appAttemptId, RMAppAttemptEventType.CONTAINER_ALLOCATED)
> container.eventHandler.handle(e)
```

显然,程序在执行 ContainerStartedTransition.transition() 的过程中又回过头来向相应 RMAppAttemptImpl 对象的状态机发出一个 CONTAINER_ALLOCATED 事件。这样,两个(或更多)状态机之间就构成一种类似于“共行程序(Co-Routine)”那样协同前进的关系,这跟两个进程或线程的协同前进本质上是一样的。

我们在前面看到,那个 RMAppAttemptImpl 状态机的当前状态为 SCHEDULED,对此事件的反应则为:

```
addTransition(RMAppAttemptState.SCHEDULED,
              EnumSet.of(RMAppAttemptState.ALLOCATED_SAVING,
                          RMAppAttemptState.SCHEDULED),
              RMAppAttemptEventType.CONTAINER_ALLOCATED,
              new AMContainerAllocatedTransition())
```

这是个多弧跳变,调用 AMContainerAllocatedTransition.transition() 的返回值将决定具体变成什么状态。

```
[FifoScheduler.assignContainer() => RMContainerImpl.ContainerStartedTransition.transition()
=> RMAppAttemptImpl.AMContainerAllocatedTransition.transition()]
```

```

AMContainerAllocatedTransition.transition(RMAppAttemptImpl appAttempt,
                                           RMAppAttemptEvent event)
> Allocation amContainerAllocation = appAttempt.scheduler.allocate(
    appAttempt.applicationAttemptId,
    EMPTY_CONTAINER_REQUEST_LIST,
    EMPTY_CONTAINER_RELEASE_LIST, null, null)
    //如果上一次在 ScheduleTransition.transition() 中对 scheduler.allocate() 的
    //调用未能达到目的,那么这一次也许可以如愿了
> if (amContainerAllocation.getContainers().size() == 0) {
    //要是还不行,就创建一个线程,让它过一会就试一下
> + appAttempt.retryFetchingAMContainer(appAttempt)
> + return RMAppAttemptState.SCHEDULED //没有容器,继续留在 SCHEDULED 状态
> }
    // amContainerAllocation 中至少有一个容器,可以往前走了
> appAttempt.setMasterContainer(amContainerAllocation.getContainers().get(0))
> rmMasterContainer = (RMContainerImpl)
    appAttempt.scheduler.getRMContainer(appAttempt.getMasterContainer().getId())
> rmMasterContainer.setAMContainer(true)
> appAttempt.rmContext.getNMTOKENSecretManager().clearNodeSetForAttempt(
    appAttempt.applicationAttemptId)
> appAttempt.getSubmissionContext().setResource(
    appAttempt.getMasterContainer().getResource())
> appAttempt.storeAttempt()
>> rmContext.getStateStore().storeNewApplicationAttempt(this)
>>> AggregateAppResourceUsage resUsage =
    appAttempt.getRMAppAttemptMetrics().getAggregateAppResourceUsage()
>>> attemptState = new ApplicationAttemptState(
    appAttempt.getAppAttemptId(), appAttempt.getMasterContainer(),
    credentials, appAttempt.getStartTime(), resUsage.getMemorySeconds(),
    resUsage.getVcoreSeconds())
>>> e = new RMStateStoreAppAttemptEvent(attemptState)
>>>> super(RMStateStoreEventType.STORE_APP_ATTEMPT)
>>>> this.attemptState = attemptState
>>> dispatcher.getEventHandler().handle(e) //发出并处理 STORE_APP_ATTEMPT 事件
> return RMAppAttemptState.ALLOCATED_SAVING //appAttempt 的状态机向前走了一步

```

我们在前面看到,当一个 AppAttempt 还处于 SUBMITTED 状态,并受到 ATTEMPT_ADDED 事件的触发时,会执行 ScheduleTransition.transition() 并从中调用 scheduler.allocate(), 企图从其自身的 newlyAllocatedContainers 集合中收揽已经分配的容器,然后转入 SCHEDULED 状态。同时,每收揽到一个容器就会向该容器发出一个 ACQUIRED 事件。正是这个 ACQUIRED 会使整个流程继续往前走,要不然这 AppAttempt 就会停滞在 SCHEDULED 状

态。这里的道理很简单:AppAttempt 拿不到资源,没有容器,当然就无法前行。但是,由于某个或某些节点上的变化,现在资源可能有了,可能已经有容器存在于这个 AppAttempt 的 newlyAllocatedContainers 集合中,所以要让这个 AppAttempt 再试一下。于是,在 CONTAINER_ALLOCATED 事件的触发下,这里再次调用 appAttempt.scheduler.allocate(),让它再试一下。要是还失败,就干脆创建一个线程,让它过一会儿就试一下,这就是 retryFetchingAMContainer()的作用:

```
[FifoScheduler.assignContainer() => RMContainerImpl.ContainerStartedTransition.transition()
=> RMAppAttemptImpl.AMContainerAllocatedTransition.transition()
> RMAppAttemptImpl.retryFetchingAMContainer()]
```

```
private void retryFetchingAMContainer(final RMAppAttemptImpl appAttempt) {
    // start a new thread so that we are not blocking main dispatcher thread.
    new Thread() {
        @Override
        public void run() {    //线程的 run()函数
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                LOG.warn("Interrupted while waiting to resend the"
                    + " ContainerAllocated Event.");
            }
            appAttempt.eventHandler.handle(new RMAppAttemptContainerAllocatedEvent(
                appAttempt.applicationAttemptId)); //再发一次 CONTAINER_ALLOCATED
                                                    //使再次调用 scheduler.allocate()
        }
    }.start(); //启动这个线程
}
```

至于这个 appAttempt,在 SCHEDULED 状态下受 CONTAINER_ALLOCATED 事件的触发,如果拿不到所需的容器,就还是停留在 SCHEDULED 状态。但是,如果拿到了一个容器,那就足可用它来发起一个 AppMaster,成立这个 App 的“项目组”了,因为 AppMaster 并不需要消耗太多的资源,它所做的是项目管理。所以,appAttempt 拿到的第一个容器称为其“主容器(MasterContainer)”。不过,在把容器发送到目标节点之前还要把这 appAttempt 保存一下,以便万一本次尝试失败可以很快重试。所以这里就调用 appAttempt.storeAttempt(),向状态保存机制发送一个 STORE_APP_ATTEMPT 事件,请求保存这个 appAttempt 的状态和相关信息。可想而知,appAttempt 必须等状态保存完成之后才能有进一步的行动。

不过这里还有个问题:如果一个 NodeManager 心跳的时候 appAttempt 尚未到来,当然这个节点的资源状况会被记录在相应的 FiCaSchedulerNode 中,但是却不会有容器进入任何 FiCaSchedulerApp 的 newlyAllocatedContainers 集合中。然后,当 appAttempt 到来并执行 scheduler.allocate()时,其 newlyAllocatedContainers 中当然是空的,但其实这时候是有资源

的,这个问题怎么解决?那就等到下一次心跳。

状态保存是由 RMStateStore 实现的,不过 RMStateStore 是个抽象类,具体加以落实的有 MemoryRMStateStore、FileSystemRMStateStore 以及 ZKRMStateStore,还有类似于/dev/null 那样的 NullRMStateStore。与 RMStateStore 相关的代码,我们就不深入进去了,有兴趣的读者可以自行阅读分析。这里只要知道,完成了状态保存之后,RMStateStore 会向作为“物主”的 appAttempt 发送一个 ATTEMPT_NEW_SAVED 事件。而 appAttempt 的状态机对此的反应则是:

```
addTransition(RMAppAttemptState.ALLOCATED_SAVING,
              RMAppAttemptState.ALLOCATED,
              RMAppAttemptEventType.ATTEMPT_NEW_SAVED,
              new AttemptStoredTransition())
```

就是说,先调用 AttemptStoredTransition.transition(),然后进入 ALLOCATED 状态。

```
AttemptStoredTransition.transition(RMAppAttemptImpl appAttempt,
                                   RMAppAttemptEvent event)

> appAttempt.launchAttempt() == RMAppAttemptImpl.launchAttempt()
>> eventHandler.handle(new AMLauncherEvent(AMLauncherEventType.LAUNCH, this))
```

这里唯一的操作就是发出 AMLauncherEventType.LAUNCH 事件,实际上相当于命令。类型为 AMLauncherEventType 的事件的受主是 ApplicationMasterLauncher,这是 RM 内部的一个成分。

ResourceManager 在其初始化阶段的 serviceInit()中通过 createAMLancher()创建了一个 ApplicationMasterLauncher 对象,此后就一直在岗备用。顾名思义,这个对象的职责就是根据具体 RMAppAttemptImpl 的要求发起 ApplicationMaster,其 handle()函数的摘要为:

```
[RMAppAttemptEventType.ATTEMPT_NEW_SAVED => AttemptStoredTransition.transition()
> ApplicationMasterLauncher.handle()]

ApplicationMasterLauncher.handle(AMLauncherEvent appEvent)
> AMLauncherEventType event = appEvent.getType()
> RMAppAttempt application = appEvent.getAppAttempt()
> switch (event) {
> case LAUNCH:
>+ launch(application) //对于 LAUNCH 命令的反应就是 launch()
>+> Runnable launcher =
               createRunnableLauncher(application, AMLauncherEventType.LAUNCH)
>+>> Runnable launcher = new AMLauncher(context, application, event, getConfig())
>+>> return launcher
>+> masterEvents.add(launcher) //挂入 ApplicationMasterLauncher 的等待队列
> case CLEANUP:
>+ cleanup(application)
```

```
> }
```

对于具体类型为 LAUNCH 的 AMLauncherEvent 事件, ApplicationMasterLauncher 执行其 launch() 方法, 通过 createRunnableLauncher() 为具体的 appAttempt 创建一个作为 Runnable 的 AMLauncher 对象, 准备作为一个线程让其专管具体 ApplicationMaster 即 AM 的发起。这个线程的作用就像个发射架。对此我们需要结合 ApplicationMasterLauncher 类的全貌才能更好地理解:

```
class ApplicationMasterLauncher extends AbstractService
    implements EventHandler<AMLauncherEvent> {}

] ThreadPoolExecutor launcherPool           //一个线程池
] LauncherThread launcherHandlingThread     //内部定义的一个类,是线程,见后
] BlockingQueue<Runnable> masterEvents = new LinkedBlockingQueue<Runnable>()
] RMContext context
] ApplicationMasterLauncher(RMContext context)
    > this.context = context
    > this.launcherPool = new ThreadPoolExecutor(10, 10, 1, TimeUnit.HOURS,
                                                new LinkedBlockingQueue<Runnable>())
        //创建 launcherPool 线程池及其内部的阻塞式队列
    > this.launcherHandlingThread = new LauncherThread()
        //创建 launcherHandlingThread 线程
] serviceStart()
    > launcherHandlingThread.start() //启动 launcherHandlingThread 线程
    > super.serviceStart()
] launch(RMAppAttempt application)
] class LauncherThread extends Thread {} //内部定义的一个类,是对线程的扩充
]] run()
    > while (!this.isInterrupted()) {
    >+ Runnable toLaunch = masterEvents.take()
        //从队列中取出,这里的 toLaunch 就是前面挂入队列的 AMLauncher
    >+ launcherPool.execute(toLaunch)
        //将线程池中的一个线程用于 AMLauncher,执行 AMLauncher.run()
    > }
```

RM 一开始就创建了这么一个对象,其内部有个线程池 launcherPool,是一组预先创建并可周转使用的线程。这些线程并不固定与具体的可执行程序绑定,而是可以通过例如 launcherPool.execute() 从池中取一线程临时绑定一个 Runnable 以执行其代码,执行完了就退出 Runnable,但线程仍旧存在,只是不再活动,直到下次又与另一个 Runnable 临时绑定。这样,就不必每当需要创建线程时就来临时创建,可以提高效率。

这里的线程池大小为 10。这是因为,系统中完全可能有多个 AppAttempt 同时需要发起各自的 AM,所以只用一个“发射架”显然是不够的;但是一个具体的 AM 一旦发起成功以后就不再需要有这么个线程了,所以需要同时存在的“发射架”数量也无须太大,线程池的大小为

10 也就可以了。另外,其内部还有个队列 `masterEvents`,这是个 `Runnable` 的队列,是用来让发射物即具体的 `AMLancher` 在此排队等待有线程来执行这些 `Runnable` 的。最后是线程 `launcherHandlingThread`,其类型就是这里定义的 `LauncherThread`。这个线程的 `run()` 函数就是一个简单的 `while` 循环,它总是盯着 `masterEvents` 这个队列,里面没有东西就睡眠,一旦有了 `AMLancher` 对象就从线程池里找一个空闲的线程,使其执行该 `AMLancher` 对象的 `run()` 函数。

下面就是 `AMLancher` 线程的事了。

事实上,这个 `AMLancher` 线程将把一个包含着主容器的 `ContainerLaunchContext` 发送到所指派的某个节点上,由那个节点上的 `NodeManager` 在其宿主机上发起一个 JVM 进程,以执行 `ContainerLaunchContext` 中给定的命令行。而 `ContainerLaunchContext` 中的这个命令行,则来自客户端提交的 `ApplicationSubmissionContext`。

第 7 章

NodeManager 与任务投运

用户提交的作业为 ResourceManager 接受并得到调度运行之后, RM 会设法将其投入运行。但是一个作业(Job 或 App)通常都包含着很多任务,比方说 N 个 MapTask 和 1 个 ReduceTask,所以作业的投运终究会分解成许多任务的投运。

为了投运一个作业, RM 首先得在某个 NodeManager 节点上启动一个进程作为这个作业的主管,扮演类似于项目组长那样的角色,这就是 ApplicationMaster 即 AM,对于 MapReduce 作业就是 MRAppMaster。后者是前者的一种特例,其 Application 是专门进行 MapReduce 计算的 App。然而,虽然这个 App 本身是多任务的作业,其 ApplicationMaster 或 MRAppMaster 却只是单任务的作业,所以 ApplicationMaster 或 MRAppMaster 的投运其实只相当于一个单任务的投运。而且, RM 之于 ApplicationMaster 的投运又恰如 ApplicationMaster 之于具体计算任务(如 MapTask)的投运,其作用和过程都很相似。

这样,如果我们顺着作业从提交到得到运行的流程,则势必先要走过任务投运的过程,而任务的投运实际上就是容器的投运,作业的投运则寓于容器投运之中。所以,本章所述的任务投运过程,既适用于 RM 对 ApplicationMaster 或 MRAppMaster 的投运,也适用于 ApplicationMaster 或 MRAppMaster 对具体的单个任务如 MapTask 等的投运。

7.1 AMLauncher 与任务投运

我们在前一章中看到,把一个作业,或者说一个 App 提交给 RM 及其调度器之后,如果不考虑与调度策略有关的细节,就像 FifoScheduler 那样只是先来先服务,那么大致的流程就是:先为其创建一个 RMApAttemptImpl,代表着运行这个 App 的一次尝试,再为其分配一个“主容器”,代表着某个 NM 节点上供其发起一个 ApplicationMaster 进程所需的一组资源,然后就为其创建一个 AMLauncher 线程,由这个线程把包含着这个主容器的一个 ContainerLaunchContext 对象“投射”到容器所属的 NM 节点上去。

下面是 AMLauncher 类的摘要:

```
class AMLauncher implements Runnable {}  
] ContainerManagementProtocol containerMgrProxy  
] RMAppAttempt application  
] Configuration conf  
] RMContext rmContext  
] Container masterContainer
```

```

] EventHandler handler
] AMLauncher(RMContext rmContext, RMAppAttempt application,
               AMLauncherEventType eventType, Configuration conf)
               //构造函数,参数 eventType 为 LAUNCH 或 CLEANUP

] connect()
  > masterContainerID = masterContainer.getId()
  > containerMgrProxy = getContainerMgrProxy(masterContainerID)
] launch()
  > connect()
  > ...
] cleanup()
  > connect()
  > ...
] run()

```

如前所述,AMLauncher 是个 Runnable。Runnable 在刚创建的时候只是个一般的对象,JVM 不会直接就使 Runnable 对象作为线程运行,而要到程序中由某个线程要求 execute 这个 Runnable 对象时才调度其运行,使其 run()函数得到调用。

AMLauncher.run()

```

> switch(eventType){ //eventType 是在创建这个 AMLauncher 对象时作为参数传下来的
> case LAUNCH:
>+ launch() == AMLauncher.launch() //执行 AMLauncher.launch()
>+ handler.handle(new RMAppAttemptEvent(
               application.getAppAttemptId(), RMAppAttemptEventType.LAUNCHED))
> case CLEANUP:
>+ cleanup() //AMLauncher.cleanup()
> }

```

如果以 LAUNCH 为创建 AMLauncher 对象时的参数 eventType,这里就执行 case LAUNCH。下次如果改成以 CLEANUP 为创建 AMLauncher 的参数,就执行 case CLEANUP。

注意,这个 run()函数里面并没有循环,而只是一个 switch 语句,执行完这个 switch 语句就返回了,这个 Runnable 与线程的绑定也就结束了。虽然线程池中的线程物理上仍旧存在,但是逻辑上可以认为这个 AMLauncher 线程已不复存在。

在我们这个情景中,事件类型 eventType 是 AMLauncherEventType.LAUNCH,所以这里只是做了两件事:其一是通过 AMLauncher.launch()发起 AM 在远地即某个 NM 结点上的创建和运行;其二是向本地即 RM 结点上的 AppAttempt 对象发出一个 LAUNCHED 事件,使其状态机进行后续的操作并发生相应的跳变。下面我们分别加以考察。

先看如何发起 ApplicationMaster 进程,即 AM 在 NM 节点上的创建和运行。

```
[AMLauncher.run() > launch()]
```

AMLauncher.launch()

```

> connect() //这是 AMLauncher.connect()
> masterContainerID = masterContainer.getId()
> ApplicationSubmissionContext applicationContext = application.getSubmissionContext()
//ApplicationSubmissionContext 来自用户的作业提交
> ContainerLaunchContext launchContext =
    createAMContainerLaunchContext(applicationContext, masterContainerID)
//ContainerLaunchContext 用于让 NM 节点创建进程以执行任务
> StartContainerRequest scRequest = StartContainerRequest.newInstance(
    launchContext, masterContainer.getContainerToken())
> List<StartContainerRequest> list = new ArrayList<StartContainerRequest>()
> list.add(scRequest) //把 StartContainerRequest 对象加到 List 中
> StartContainersRequest allRequests = StartContainersRequest.newInstance(list)
//真正发往 NM 节点的是一个 List,但是这一次的 List 中只有一个请求
> StartContainersResponse response = containerMgrProxy.startContainers(allRequests)
== ContainerManagementProtocol.startContainers(allRequests)

```

AMLauncher 在 RM 节点上,而要创建的 ApplicationMaster 进程是在某个 NM 节点上,这是跨节点的操作,所以要用到 Hadoop 的 RPC 机制。这里的 connect()就是与对方建立 RPC 连接的过程,其主体 getContainerMgrProxy()返回一个 containerMgrProxy,这个 Proxy 通过 ProtoBuf 实现了与对岸 ContainerManagementProtocol 界面上的 RPC 连接,使得 RM 节点上对此 Proxy 所提供函数的调用原封不动地转化成对岸对于相同界面的函数调用。值得注意的是,在集群中 RM 是主节点,总是起着服务方的作用;而 NM 所在的都是从节点,都是主动发起对 RM 节点通信,是要求得到服务的客户方。但是,在 AMLauncher 与 NM 节点的交互中却倒过来了,现在是 AMLauncher 在请求 NM 提供服务,AMLauncher 扮演着客户方的角色,而 NM 扮演着服务方的角色。

```
[AMLauncher.run() > launch() > connect()]
```

AMLauncher.connect()

```

> ContainerId masterContainerID = masterContainer.getId()
> containerMgrProxy = getContainerMgrProxy(masterContainerID)
>> NodeId node = masterContainer.getNodeId() //主容器中有对方的 NodeId
>> InetAddress containerManagerBindAddress = //还有对方的 IP 地址和端口号
    NetUtils.createSocketAddrForHost(node.getHost(), node.getPort())
>> YarnRPC rpc = YarnRPC.create(conf) //创建底层 RPC
>> UserGroupInformation currentUser = //创建一个 UGI
    UserGroupInformation.createRemoteUser(
        containerId.getApplicationAttemptId().toString())
>> String user = rmContext.getRMApps()
    .get(containerId.getApplicationAttemptId().getApplicationId()).getUser()

```



```
>> token = rmContext.getNMTTokenSecretManager().createNMTToken(
    containerId.getApplicationAttemptId(), node, user)
    //创建用于对方节点的证章 Token
>> currentUser.addToken(ConverterUtils.convertFromYarn( //把证章加到 UGI 中
    token, containerManagerBindAddress))
>> pa = new PrivilegedAction<ContainerManagementProtocol>()
    ] ContainerManagementProtocol run()
    > return (ContainerManagementProtocol) rpc.getProxy(
    ContainerManagementProtocol.class, containerManagerBindAddress, conf)
>> return currentUser.doAs(pa) //以此 user 的名义执行 PrivilegedAction 对象的 run()函数
```

这个函数首先与对方建立起 Socket 层的连接,并为用户创建 UGI 和 Token,然后以这个用户的名义执行这里动态定义的 PrivilegedAction 对象中的 run()函数,以获取或创建代表着对方的 Proxy。当然,这里实际上蕴含着许多底层操作,读者可以参阅本书第 4 章。有关 UGI 和 Token 则可参阅本书第 17 章。

这个 connect()不光是在 launch()中要调用,在 cleanup()中同样也要调用。因为由 AMLauncher 建立的 RPC 通道并不需要持续存在,而只是在 LAUNCH 或 CLEANUP 的时候临时用一下,通常 LAUNCH 与 CLEANUP 之间的时间间隔是不小的。

与对方建立起 RPC 通道之后,launch()通过 Proxy 跨节点调用对方的 startContainers(),这个 Proxy 就是程序中的 containerMgrProxy,其类型为 ContainerManagementProtocol。但是那只是个界面,所以实体的类型是在 Client 一侧(即 RM 这一边)实现了这个界面的 ContainerManagementProtocolPBClientImpl。对于 RPC,跨节点进行调用的一方是 Client,被调用的一方是 Server。下面是 Client 一侧的这个函数的代码:

```
[AMLauncher.run() > launch() > ContainerManagementProtocolPBClientImpl.startContainers()]

public StartContainersResponse
    startContainers(StartContainersRequest requests) throws YarnException, IOException {
    StartContainersRequestProto requestProto =
        ((StartContainersRequestPBImp) requests).getProto();
    try {
        return new StartContainersResponsePBImp(proxy.startContainers(null, requestProto));
    } catch (ServiceException e) {
        RPCUtil.unwrapAndThrowException(e);
        return null;
    }
}
```

参数 requests 是个 StartContainersRequest 对象,但那是个抽象类,实际上落实的类型是 StartContainersRequestPBImp,其 getProto()将其加工成一个 PB 报文,然后就通过下一层的 Proxy 调用对岸的同名函数。这个 Proxy 的类型为 ContainerManagementProtocolPB,但其实这是个界面,所以 proxy 是实现了这个界面的某类对象,这是由 ProtoBuf 动态定义的。而上一层

的 containerMgrProxy 的类型则是 ContainerManagementProtocolPBClientImpl。这二者有层次之分,前者是 ProtoBuf 底层的 Proxy,后者则是以 ProtoBuf 为基础的 ContainerManagementProtocol 层的 Proxy。

程序中先调用 proxy.startContainers(),然后以其来自对岸的返回值为参数构建一个 StartContainersResponsePBImp 对象并将其返回给上一层作为响应。

而对岸执行 startContainers()的那些活动,当然就是在 NM 节点上了。

在转入 NM 节点之前,我们先看一下 RM 节点上那个 AppAttempt 状态机的操作,其相关的跳变规则是这样的:

```
addTransition(RMAppAttemptState.ALLOCATED, RMAppAttemptState.LAUNCHED,
              RMAppAttemptEventType.LAUNCHED, new AMLaunchedTransition())
```

在前述 LAUNCHED 事件的驱动下,它先执行 AMLaunchedTransition.transition(),然后就转入 LAUNCHED 状态。

```
[AMLauncher.run() => RMAppAttemptImpl.AMLaunchedTransition.transition()]
```

```
AMLaunchedTransition.transition()
```

```
> appAttempt.attemptLaunched(); //让 AMLivelinessMonitor 监视 AM 是否存活
```

```
> appAttempt.rmContext.getClientToAMTokenSecretManager().registerApplication(
    appAttempt.getAppAttemptId(), appAttempt.getClientTokenMasterKey())
    == ClientToAMTokenSecretManager.registerApplication() //属于安全机制
```

AppAttempt 状态机进入了 LAUNCHED 状态,下面就是静候来自 NM 节点的确认和报告。到了那时候,这个状态机就会转入 RUNNING 状态。

7.2 MRAppMaster 或 AM 的创建

现在,“球”滚到了 NM 节点上。

通过 RPC 和 PB 的作用,NM 节点上 ContainerManagerImpl 的 startContainers()函数受到了调用。RM 节点上对 containerMgrProxy.startContainers()的调用转化成 NM 节点上对 ContainerManagerImpl.startContainers()的调用。

其实 NM 节点上最先受到调用的是 ContainerManagementProtocolPBServiceImpl 的 startContainers():

```
ContainerManagementProtocolPBServiceImpl.startContainers(
    RpcController arg0, StartContainersRequestProto proto)
> StartContainersRequestPBImp request = new StartContainersRequestPBImp(proto)
    //根据请求报文构建 StartContainersRequestPBImp 对象
> StartContainersResponse response = real.startContainers(request) //调用 startContainers()
    == ContainerManagerImpl.startContainers(StartContainersRequest requests)
> return ((StartContainersResponsePBImp)response).getProto() //这是要返回 RM 节点的报文
```

这里的 `real` 是一个实现了 `ContainerManagementProtocol` 界面的对象, 实际上就是个 `ContainerManagerImpl` 对象, 这才是真正实现了界面所定义操作的对象。我们在这里并不关心 RPC 和 PB 的细节, 所以就直接从 `ContainerManagerImpl.startContainers()` 开始。

```
[ContainerManagementProtocolPBServiceImpl.startContainers() >
ContainerManagerImpl.startContainers()]

class ContainerManagerImpl extends CompositeService implements ...{}

] startContainers(StartContainersRequest requests)
    //Start a list of containers on this NodeManager.
    > nmTokenIdentifier = selectNMTokenIdentifier(remoteUgi)
    > authorizeUser(remoteUgi, nmTokenIdentifier)
    > for (StartContainerRequest request : requests.getStartContainerRequests()) {
    >+ containerTokenIdentifier = BuilderUtils.newContainerTokenIdentifier(
                                                request.getContainerToken())
    >+ containerId = containerTokenIdentifier.getContainerID()
    >+ startContainerInternal(nmTokenIdentifier, containerTokenIdentifier, request) // 见下
    >+ succeededContainers.add(containerId) // 如果没有发生异常, 那就是成功了
    >+ catch (YarnException e)
    >+ failedContainers.put(containerId, ...) // 如果过程中发生异常, 那就是启动失败
    > } //end for, 对于请求中列举的所有容器
    > return StartContainersResponse.newInstance(getAuxServiceMetaData(),
                                                succeededContainers, failedContainers)
    //在返回的响应报文中将分别列举成功和失败的容器

] startContainerInternal(NMTokenIdentifier nmTokenIdentifier, ...)
    > updateNMTokenIdentifier(nmTokenIdentifier)
    > containerId = containerTokenIdentifier.getContainerID()
    > user = containerTokenIdentifier.getApplicationSubmitter()
    > LOG.info("Start request for " + containerIdStr + " by user " + user)
    > ContainerLaunchContext launchContext = request.getContainerLaunchContext()
    //从 request 中恢复出 ContainerLaunchContext, 即 CLC
    > serviceData = getAuxServiceMetaData()
    > //创建一个 ContainerImpl
    > Container container = new ContainerImpl(getConfig(), this.dispatcher,
                                            context.getNMStateStore(), launchContext, ...)
    //在 NM 节点上创建一个 ContainerImpl 对象
    > ApplicationId applicationID = containerId.getApplicationAttemptId().getApplicationId()
    > Application application = new ApplicationImpl(dispatcher, user,
                                                    applicationID, credentials, context) //创建一个 ApplicationImpl
    //注意这个 dispatcher 就是 ContainerManagerImpl.dispatcher
    > if (null == context.getApplications().putIfAbsent(applicationID, application)) {
```

```

>+ LOG.info("Creating a new application reference for app " + applicationID)
>+ e = new ApplicationInitEvent(applicationID, appAcls, logAggregationContext)
>+> super(appId, ApplicationEventType.INIT_APPLICATION)
//产生一个 INIT_APPLICATION 事件
>+ dispatcher.getEventHandler().handle(e) //用来触发这个 ApplicationImpl 的状态机
> }
> e = new ApplicationContainerInitEvent(container) //用来运载这个新建的 Container
>> super(..., ApplicationEventType.INIT_CONTAINER)
//产生一个 ApplicationEventType.INIT_CONTAINER 事件
> dispatcher.getEventHandler().handle(e) //把 Container 交给 ApplicationImpl
> metrics.launchedContainer() //用于统计目的
> metrics.allocateContainer(containerTokenIdentifier.getResource()) //用于统计目的

```

这里创建了两个对象：一个是 ApplicationImpl 对象，这是具体 App 在这 NM 节点上的代表；另一个是 ContainerImpl 对象，在 NM 节点上代表着容器（在 RM 节点上是 RMContainerImpl 对象），实质上代表着已经指派给将要在这 NM 节点上投运的程序的那部分资源。然后一方面向这个 ApplicationImpl 发送 INIT_APPLICATION 事件，触发其状态机；一方面又将刚创建的 ContainerImpl 对象搭载在一个 ApplicationEventType.INIT_CONTAINER 事件上，也将其发送给这个 ApplicationImpl 对象。所以是向这个 ApplicationImpl 对象先后发出两个事件。

先看前者，ApplicationImpl 对 INIT_APPLICATION 事件的反应是：

```

addTransition(ApplicationState.NEW, ApplicationState.INITING,
               ApplicationEventType.INIT_APPLICATION, new AppInitTransition())

```

即执行 AppInitTransition.transition()，然后发生从 NEW 到 INITING 的状态跳变。

```

[ContainerManagerImpl.startContainers() > startContainerInternal()
=> ApplicationImpl.AppInitTransition.transition()]

ApplicationImpl.AppInitTransition.transition(ApplicationImpl app, ApplicationEvent event)
> initEvent = (ApplicationInitEvent)event
> app.applicationACLs = initEvent.getApplicationACLs()
> app.aclsManager.addApplication(app.getAppId(), app.applicationACLs)
> app.logAggregationContext = initEvent.getLogAggregationContext()
> e = new LogHandlerAppStartedEvent(app.getAppId(),...
//产生一个 LogHandlerAppStartedEvent 事件
//ContainerLogsRetentionPolicy.ALL_CONTAINERS, ...)
>> super(LogHandlerEventType.APPLICATION_STARTED) //表面上看似只是用于日志的事件
> app.dispatcher.getEventHandler().handle(e) == LogAggregationService.handle()
//触发日志服务的状态机，启动具体 App 的运行日志(LOG)机制

```

NM 结点上 App 的初始化阶段，第一件大事就是启动其运行日志(LOG)机制。一个 NM 节点上可能有好多 App 在运行，有好多的容器，这些容器在投运的过程中都会产生日志信息，

但是 NM 节点需要将这些 LOG 信息妥善合并(Aggregate)在一起。不过这个 LOG 机制可以启用,也可以不启用,取决于配置文件 yarn-default.xml 中的配置项“yarn.log-aggregation-enable”的值为 true 或 false。如果启用,ContainerManager 就会为此创建一个 LogAggregationService 对象来专门处理这个事情,否则就会创建一个 NonAggregatingLogHandler。这里我们假定采用默认值 false,那就不启用 LOG 机制,于是这里的 EventHandler 就是 NonAggregatingLogHandler。

不过我们在这里对 LOG 机制本身不感兴趣,所以跳过相关的细节,但是需要知道:不管其为 LogAggregationService 还是 NonAggregatingLogHandler,当具体 App 完成其 LOG 机制的初始化时,都会对其状态机发出一个 APPLICATION_LOG_HANDLING_INITED 事件,ApplicationImpl 对象的状态机会因此而调用 AppLogInitDoneTransition.transition(),但维持 INITING 状态不变。后面我们将看到,正是这一步操作开启了 AM 的资源本地化过程。

另一个事件 ApplicationEventType.INIT_CONTAINER 也是针对 ApplicationImpl 的。注意,这是针对 ApplicationImpl 的 ApplicationEventType.INIT_CONTAINER,而不是针对 ContainerImpl 的 ContainerEventType.INIT_CONTAINER,这是两种不同的事件。由于前一个事件 INIT_APPLICATION 的作用,ApplicationImpl 的状态已变成 INITING,而 APPLICATION_LOG_HANDLING_INITED 事件不会改变其状态,所以此时的跳变规则为:

```
addTransition(ApplicationState.INITING, ApplicationState.INITING,
               ApplicationEventType.INIT_CONTAINER,
               new InitContainerTransition())
```

即执行 InitContainerTransition.transition()但仍维持状态 INITING 不变。

```
InitContainerTransition.transition(ApplicationImpl app, ApplicationEvent event)
> initEvent = (ApplicationContainerInitEvent) event
> Container container = initEvent.getContainer()
> app.containers.put(container.getContainerId(), container)
    //将 container 放在这个 ApplicationImpl 的 containers 集合中
> LOG.info("Adding " + container.getContainerId() + " to application " + app.toString())
> switch (app.getApplicationState()) {
> case INITING;
> case NEW;
>+ break //these get queued up and sent out in AppInitDoneTransition
> }
```

可见这只是将运载在 INIT_CONTAINER 事件上的容器放在 ApplicationImpl 对象的 containers 集合中而已。

然而,如前所述,对于 LOG 机制的初始化完成时,不管其为 LogAggregationService 或 NonAggregatingLogHandler,都会发出一个 APPLICATION_LOG_HANDLING_INITED 事件,使 ApplicationImpl 对象的状态机调用 AppLogInitDoneTransition.transition():

```
[ContainerManagerImpl.startContainers() > startContainerInternal()
=> ApplicationImpl.AppInitTransition.transition()
=> ApplicationImpl.AppLogInitDoneTransition().transition()]
```

```
AppLogInitDoneTransition().transition()
> e = new ApplicationLocalizationEvent(
    LocalizationEventType.INIT_APPLICATION_RESOURCES, app)
> app.dispatcher.getEventHandler().handle(e)
```

这里产生了一个 LocalizationEventType.INIT_APPLICATION_RESOURCES 事件,并用此事件去触发某个状态机。注意,这是在 NodeManager 节点上,这里的 app.dispatcher 就是 ContainerManagerImpl 在前面 startContainerInternal() 创建这个 ApplicationImpl 对象时传下的 ContainerManagerImpl.dispatcher,这也是个 AsyncDispatcher。向这个 dispatcher 登记接受 LocalizationEventType 事件的是 ResourceLocalizationService。

而 ResourceLocalizationService 对这个事件的反应是 handleInitApplicationResources():

```
[ContainerManagerImpl.startContainers() > startContainerInternal()
=> ApplicationImpl.AppInitTransition.transition()
=> ApplicationImpl.AppLogInitDoneTransition().transition()
=> ResourceLocalizationService.handle()]
```

```
public void handle(LocalizationEvent event) {
    // TODO: create log dir as $logdir/$user/$appId
    switch (event.getType()) {
        case INIT_APPLICATION_RESOURCES:
            handleInitApplicationResources(
                ((ApplicationLocalizationEvent)event).getApplication());
            break;
        case INIT_CONTAINER_RESOURCES:
            handleInitContainerResources((ContainerLocalizationRequestEvent) event);
            break;
        case CACHE_CLEANUP:
            handleCacheCleanup(event);
            break;
        case CLEANUP_CONTAINER_RESOURCES:
            handleCleanupContainerResources((ContainerLocalizationCleanupEvent)event);
            break;
        case DESTROY_APPLICATION_RESOURCES:
            handleDestroyApplicationResources(
                ((ApplicationLocalizationEvent)event).getApplication());
            break;
        default:
            throw new YarnRuntimeException("Unknown localization event: " + event);
    }
}
```



```
}
```

对于 INIT_APPLICATION_RESOURCES, 这里调用 handleInitApplicationResources(), 下面是这个函数的代码:

```
[ContainerManagerImpl.startContainers() > startContainerInternal()
=> ApplicationImpl.AppInitTransition.transition()
=> ApplicationImp.AppLogInitDoneTransition().transition()
=> ResourceLocalizationService.handle() > handleInitApplicationResources()]
```

```
private void handleInitApplicationResources(Application app) {
    // 0) Create application tracking structs
    String userName = app.getUser();
    privateRsrc.putIfAbsent(userName, new LocalResourcesTrackerImpl(
        userName, null, dispatcher, true, super.getConfig(), stateStore));
    // 创建一个 LocalResourcesTrackerImpl 对象, 并将其放在 privateRsrc 集合中
    String appIdStr = ConverterUtils.toString(app.getAppId());
    appRsrc.putIfAbsent(appIdStr, new LocalResourcesTrackerImpl(
        app.getUser(), app.getAppId(), dispatcher, false, super.getConfig(), stateStore));
    // 再创建一个 LocalResourcesTrackerImpl 对象, 并将其放在 appRsrc 集合中
    // 1) Signal container init
    // This is handled by the ApplicationImpl state machine and allows
    // containers to proceed with launching.
    dispatcher.getEventHandler().handle(new ApplicationInitEvent(app.getAppId()));
    // 实际上是 ApplicationEventType.APPLICATION_INITED
}
```

这里创建了两个 LocalResourcesTrackerImpl 对象, 一个放在以用户名为键的集合 privateRsrc 中; 另一个放在以 appIdStr 即应用名为键的集合 appRsrc 中。这两个对象还有个不同之处, 就是在创建对象时的第四个参数, 一个为 true, 另一个为 false。这是什么意思呢? LocalResourcesTrackerImpl 构造函数的第四个参数是 useLocalCacheDirectoryManager, 意为是否使用 LocalCacheDirectoryManager, 即本地的缓存目录管理。

然后, 就向具体 ApplicationImpl 的状态机发出一个 ApplicationInitEvent 事件, 实际上这是 ApplicationEventType.APPLICATION_INITED。ApplicationImpl 状态机中相应的跳变规则为:

```
addTransition(ApplicationState.INITING, ApplicationState.RUNNING,
    ApplicationEventType.APPLICATION_INITED, new AppInitDoneTransition())
```

受 APPLICATION_INITED 事件的触发, 状态机执行 AppInitDoneTransition.transition(), 然后从 INITING 状态转入 RUNNING 状态。

下面是 AppInitDoneTransition.transition() 的摘要:

```
[ApplicationImp.AppLogInitDoneTransition()
```

```
=> ResourceLocalizationService.handle() > handleInitApplicationResources()
=> ApplicationImpl.AppInitDoneTransition.transition()]
```

```
ApplicationImpl.AppInitDoneTransition.transition()
> for (Container container : app.containers.values()) { //对于 containers 集合中的每个容器
>+ e = new ContainerInitEvent(container.getId())
>+> super(c, ContainerEventType.INIT_CONTAINER) //产生一个 INIT_CONTAINER 事件
>+ app.dispatcher.getEventHandler().handle(e) //驱动该容器的状态机
> } //end for
```

针对 App 自身的初始化已经完成,下面该是针对容器的初始化了,所以对其 containers 集合中的每一个容器发出一个 INIT_CONTAINER 事件,让它们进行初始化。之所以这里有个 for 循环,是因为在同一个 NM 节点上指派给同一个 App 的容器可能不止一个,因为这节点上可能会运行这个 App 的多个 Mapper,还可能又会运行其 Reducer。注意,这次产生的是 ContainerEventType.INIT_CONTAINER (而不是 ApplicationEventType.INIT_CONTAINER),这当然是针对具体容器,即 ContainerImpl 对象的状态机的。而 dispatcher,虽然是同一个 dispatcher,却会根据事件的类型和预先登记的路由加以分发,所以这里就到了具体的容器即 ContainerImpl 对象那里。注意,RM 节点上代表着容器的是 RMContainerImpl,而在 NM 节点上则是 ContainerImpl 对象。ContainerImpl 状态机对此事件的反应则为:

```
addTransition(ContainerState.NEW,
    EnumSet.of(ContainerState.LOCALIZING,
        ContainerState.LOCALIZED,
        ContainerState.LOCALIZATION_FAILED,
        ContainerState.DONE),
    ContainerEventType.INIT_CONTAINER, new RequestResourcesTransition())
```

先执行 RequestResourcesTransition.transition(),然后从初始的状态 ContainerState.NEW 视情况的不同而转入 ContainerState.LOCALIZING 或 ContainerState.LOCALIZED 状态,或者在恢复(以前所保存的)容器的情况下也可能转入 ContainerState.DONE 状态。当然也可能因资源本地化操作失败(发生异常)而转入 LOCALIZATION_FAILED。

```
[ApplicationImp.AppLogInitDoneTransition()
=> ResourceLocalizationService.handle() > handleInitApplicationResources()
=> ApplicationImpl.AppInitDoneTransition.transition()
=> ContainerImpl.RequestResourcesTransition.transition()]
```

```
RequestResourcesTransition.transition()
> if (container.recoveredStatus == RecoveredContainerStatus.COMPLETED) {
    //容器的恢复已经完成(以前保存了容器内容,但是执行失败,需要重执)
>+ container.sendFinishedEvents() //发送 ApplicationContainerFinishedEvent 等事件
>+ return ContainerState.DONE
```

```

> } else if (container.recoveredAsKilled &&
    container.recoveredStatus == RecoveredContainerStatus.REQUESTED) {
>+ // container was killed but never launched,要求恢复的容器已被 kill
>+ container.finished() //资源本地化已完成
>+ return ContainerState.DONE
> }
> ContainerLaunchContext ctxt = container.launchContext //这是容器发起上下文 CLC
> container.metrics.initingContainer() //为统计目的
> e = new AuxServicesEvent(AuxServicesEventType.CONTAINER_INIT, container)
> container.dispatcher.getEventHandler().handle(e) //向 AuxServices 发送 CONTAINER_INIT
    //AuxServices 是一些可通过配置项加以配置的辅助性服务
> Map<String,ByteBuffer> csd = ctxt.getServiceData() //从 CLC 中获取辅助服务数据 csd
> if (csd!= null) {
>+ // This can happen more than once per Application as each container may
>+ // have distinct service data. 同一应用的不同容器可能有不同的辅助服务数据
>+ for (Map.Entry<String,ByteBuffer> service : csd.entrySet()) {
>++ //为每一项辅助服务数据向 AuxServices 发送 APPLICATION_INIT 事件
>++ e = new AuxServicesEvent(AuxServicesEventType.APPLICATION_INIT, ...)
>++ container.dispatcher.getEventHandler().handle(e) == AuxServices.handle(e)
>+ } //end for
> }
> // Send requests for public, private resources,发送资源请求
> Map<String, LocalResource> cntrRsrc = ctxt.getLocalResources() //从 CLC 获取本地资源
> if (!cntrRsrc.isEmpty()) { //有资源请求
>+ for (Map.Entry<String,LocalResource> rsrc : cntrRsrc.entrySet()) {
    //对于其中的每一项资源
>++ req = new LocalResourceRequest(rsrc.getValue()) //创建 LocalResourceRequest 对象
>++ List<String> links = container.pendingResources.get(req)
>++ if (links == null) { //如果不在 pendingResources 集合中
>+++ links = new ArrayList<String>()
>+++ container.pendingResources.put(req, links) //放入这个集合
>++ }
>++ links.add(rsrc.getKey())
>++ switch (rsrc.getValue().getVisibility()) {
>++ case PUBLIC;
>+++ container.publicRsrcs.add(req) //这是公共资源
>++ case PRIVATE;
>+++ container.privateRsrcs.add(req) //这是本容器的专用资源
>++ case APPLICATION;
>+++ container.appRsrcs.add(req) //这是由本应用所属容器共享的 App 资源

```

```

> ++ }
> + } //end for
> + Map<LocalResourceVisibility, Collection<LocalResourceRequest>> req =
    new HashMap<LocalResourceVisibility, ...>()
> + if (!container.publicRsrcs.isEmpty())
> ++ req.put(LocalResourceVisibility.PUBLIC, container.publicRsrcs) //填写公共资源请求
> + if (!container.privateRsrcs.isEmpty())
> ++ req.put(LocalResourceVisibility.PRIVATE, container.privateRsrcs) //填写专用资源请求
> + if (!container.appRsrcs.isEmpty())
> ++ req.put(LocalResourceVisibility.APPLICATION, container.appRsrcs) //填写 App 资源请求
> + //将资源本地化请求作为一个事件发送给 ResourceLocalizationService
> + e = new ContainerLocalizationRequestEvent(container, req)
> +> super(LocalizationEventType.INIT_CONTAINER_RESOURCES, c)
> + container.dispatcher.getEventHandler().handle(e) == ResourceLocalizationService.handle(e)
> + return ContainerState.LOCALIZING // ContainerImpl 转入 LOCALIZING 状态
> } // end if (!cntrRsrc.isEmpty())
> else { // cntrRsrc.isEmpty(), 没有资源请求
> + container.sendLaunchEvent()
> + container.metrics.endInitingContainer()
> + return ContainerState.LOCALIZED // ContainerImpl 转入 LOCALIZED 状态
> }

```

我们在这里只关心“主旋律”，所以把注意力集中在常规的、成功的资源本地化过程。对于 CLC 即 ContainerLaunchContext 中的资源要求，这里先进行了一番分类和整理，放在按资源开放范围即 Visibility 分类 (PUBLIC、PRIVATE、APPLICATION) 的 MAP 中。然后将此 MAP 搭载在一个 LocalizationEventType.INIT_CONTAINER_RESOURCES 事件上发送出去。登记接受 LocalizationEventType 类事件的是一个 ResourceLocalizationService 对象，负责提供资源本地化服务。这个事件说是 INIT_CONTAINER_RESOURCES，其实就是进行资源本地化的请求。资源的本地化当然会有个过程，现在才刚开始，所以这个具体容器的状态变成 LOCALIZING，表示正在进行之中。

当然，如果没有资源要求，那就不用这么麻烦了，可直接进入 LOCALIZED 状态。

ResourceLocalizationService 没有状态机，对于事件的处理由其 handle() 函数直接完成：

```

[ApplicationImp. AppLogInitDoneTransition()
=> ResourceLocalizationService.handle() > handleInitApplicationResources()
=> ApplicationImpl. AppInitDoneTransition.transition()
=> ContainerImpl. RequestResourcesTransition.transition()
> ResourceLocalizationService.handle()]

```

```

public void handle(LocalizationEvent event) {

```

```

    // TODO: create log dir as $logdir/$user/$appId

```

```

switch (event.getType()) {
case INIT_APPLICATION_RESOURCES:
    handleInitApplicationResources(
        ((ApplicationLocalizationEvent)event).getApplication());
    break;
case INIT_CONTAINER_RESOURCES:
    handleInitContainerResources((ContainerLocalizationRequestEvent) event);
    break;
case CACHE_CLEANUP:
    handleCacheCleanup(event);
    break;
case CLEANUP_CONTAINER_RESOURCES:
    handleCleanupContainerResources((ContainerLocalizationCleanupEvent)event);
    break;
case DESTROY_APPLICATION_RESOURCES:
    handleDestroyApplicationResources(
        ((ApplicationLocalizationEvent)event).getApplication());
    break;
default:
    throw new YarnRuntimeException("Unknown localization event: " + event);
}
}

```

我们在这里只关心 INIT_CONTAINER_RESOURCES。ResourceLocalizationService 对此事件的反应是调用 handleInitContainerResources()。

```

[ApplicationImp.AppLogInitDoneTransition()
=> ResourceLocalizationService.handle() > handleInitApplicationResources()
=> ApplicationImpl.AppInitDoneTransition.transition()
=> ContainerImpl.RequestResourcesTransition.transition()
> ResourceLocalizationService.handle() > handleInitContainerResources()]

```

```

ResourceLocalizationService.handleInitContainerResources(
    ContainerLocalizationRequestEvent rsrcReqs)
> Container c = rsrcReqs.getContainer() //从该事件对象中获取所涉及的容器
> LoadingCache<Path,Future<FileStatus>>> statCache = /* 建立一个缓存 */
    CacheBuilder.newBuilder().build(FSDownload.createStatusCacheLoader(getConfig()))
> ctxt = new LocalizerContext(c.getUser(), c.getContainerId(), c.getCredentials(), statCache)
    //从容器中获取相关信息,连同缓存一起创建一个本地化上下文 LocalizerContext
> Map<LocalResourceVisibility, Collection<LocalResourceRequest>>> rsrcs =
    rsrcReqs.getRequestedResources() //将搭载在事件上的资源请求转移到一个 Map 中

```

```

> for (Map.Entry<LocalResourceVisibility,
    Collection<LocalResourceRequest>> e : rsrcs.entrySet()) { //对于每一项资源:
>+ LocalResourcesTracker tracker = getLocalResourcesTracker(e.getKey(), c.getUser(),
    c.getContainerId().getApplicationAttemptId().getApplicationId())
    //根据资源的开放类别,即 e.getKey(),找到相应的本地资源跟踪者:
>+> switch (visibility) { //ResourceLocalizationService 中已经有了这些本地资源跟踪者
>+> case PUBLIC:
>+>+ return publicRsrc //这是对公共资源的 LocalResourcesTrackerImpl
>+> case PRIVATE:
>+>+ return privateRsrc.get(user) //每个用户都有自己的 LocalResourcesTrackerImpl
>+> case APPLICATION:
>+>+ return appRsrc.get(ConverterUtils.toString(appId)) //每个 App 都有各自的资源跟踪者
>+> }
>+ for (LocalResourceRequest req : e.getValue()) { //对于相同可见性的每一项资源:
>++ event = new ResourceRequestEvent(req, e.getKey(), ctxt)
>++> super(resource, ResourceEventType.REQUEST)
>++ tracker.handle(event) == LocalResourcesTrackerImpl.handle(event)
>+ } //end for(LocalResourceRequest req : e.getValue())
> } //end for(Map.Entry<LocalResourceVisibility, Collection<LocalResourceRequest>> ...)

```

我们在前面已经看到,需要加以本地化的资源按其开放范围即可见性分成三种,每一种资源都可以有许多项,所以这里通过一个两层嵌套的 for 循环加以处理。第一层是对于不同可见性的循环,第二层是对具有同一种可见性的各项资源的循环。

具体 NM 节点的 ResourceLocalizationService 在此前已经创建了一些用于不同目的的 LocalResourcesTracker,即 LocalResourcesTrackerImpl 对象。其中对公共资源的跟踪者 publicRsrc 是在 ResourceLocalizationService 初始化时就创建好的,因为预期使用频率最高,而且相对而言是固定的。而 privateRsrc 和 appRsrc 是因具体用户和 App 而变的集合,取决于两个 Map 中的内容。我们在前面看到,当 ResourceLocalizationService 的 handleInitApplicationResources() 受到调用时,会为 App 的用户创建一个跟踪者并将其放入 privateRsrc 集合,同时又为这 App 创建一个跟踪者并将其放入 appRsrc 集合。

现在,当我们要处理例如 PUBLIC 资源的本地化时,就得使用公共资源的跟踪者 publicRsrc,然后通过内层的 for 循环为每一项需要加以本地化的公共资源向其跟踪者 publicRsrc 发送一个 REQUEST 事件。同理,如果要处理的是 APPLICATION 资源的本地化,就得使用 AppID 从 appRsrc 集合中找到该 App 的跟踪者,然后就每一项 APPLICATION 资源向此跟踪者发送一个 REQUEST 事件。

而具体的跟踪者,即 LocalResourcesTrackerImpl 对象,则会(在本地)为此项资源创建一个 LocalizedResource 对象,然后由此 LocalizedResource 对象完成具体资源的本地化,就是把资源拷贝到本地来。我们先看 LocalResourcesTrackerImpl 的 handle() 函数:

```

> [ContainerImpl.RequestResourcesTransition.transition()
> ResourceLocalizationService.handle() > handleInitContainerResources()

```



```

> LocalResourcesTrackerImpl.handle()]

LocalResourcesTrackerImpl.handle(event)
> LocalResourceRequest req = event.getLocalResourceRequest() //从事件中抽取资源请求
> LocalizedResource rsrc = localrsrc.get(req)
//根据该项资源的可见性找到相应的 LocalizedResource 对象
> switch (event.getType()) {
> case REQUEST:
>+ if (rsrc != null && (!isResourcePresent(rsrc))) {
>++ LOG.info("Resource " + rsrc.getLocalPath() + " is missing, localizing it again")
>++ removeResource(req)
>++ rsrc = null
>+ }
>+ if (null == rsrc) { //尚未创建 LocalizedResource
>++ rsrc = new LocalizedResource(req, dispatcher) //创建一个 LocalizedResource 对象
>++ localrsrc.put(req, rsrc)
>+ }
> case ... //别的 case 从略,包括 LOCALIZED、RELEASE、LOCALIZATION_FAILED 等
> }
> rsrc.handle(event) == LocalizedResource.handle(event) //向 LocalizedResource 转交该事件
> if (event.getType() == ResourceEventType.LOCALIZED) {
>+ if (rsrc.getLocalPath() != null) {
>++ stateStore.finishResourceLocalization(user, appId, buildLocalizedResourceProto(rsrc))
>+ }
> }

```

我们在这里只关心对于 REQUEST 事件的处理。如前所述,事件中搭载着具体的资源请求,所以这里将其摘取出来。然后根据这个资源的可见性类别找到相应的 LocalResourceRequest 对象,如果尚未创建则加以创建。手中有对于具体资源请求的 LocalizedResource 对象之后,就将事件 event 转交给它,调用其 handle() 函数。虽然对象的类型名曰 Localized,但此刻显然尚未完成该项资源的本地化。

7.3 资源本地化

LocalizedResource 是有状态机的,在初始化阶段会创建好它的状态机,初始状态为 INIT。这个状态机对于 ResourceEventType.REQUEST 事件的跳变规则为:

```

addTransition(ResourceState.INIT, ResourceState.DOWNLOADING,
    ResourceEventType.REQUEST, new FetchResourceTransition())

```

就是说,这个状态机会执行 FetchResourceTransition.transition(), 然后从 INIT 状态转入 DOWNLOADING 状态。这里我们需要回顾一下,创建 LocalizedResource 对象并将

REQUEST 事件转交给它的是 LocalResourcesTracker, 而 REQUEST 事件的源头是 ResourceLocalizationService, 这是 NodeManager 内部 ContainerManagerImpl 下面的一个成分, 是它为需要加以本地化的每一项资源发出一个 REQUEST 事件, 并且保证每一项资源都已经有了自己的 LocalizedResource 对象, 以实施该项资源的本地化, 如果没有就创建一个。

这样, 一个作业(App)在一个具体 NM 节点上有多少项资源需要本地化, 就会有多少个 LocalizedResource 对象, 当然也就有多少个这样的状态机, 实质上就是有多少个并发的资源本地化过程。现在我们考察的是其中一项, 其中的一个状态机。

不过 LocalizedResource 的 FetchResourceTransition.transition() 只是为本项资源的本地化做些简单的准备, 真正的本地化服务还是由 ResourceLocalizationService 提供。

```
[LocalizedResource.FetchResourceTransition.transition()]
```

```
FetchResourceTransition.transition(LocalizedResource rsrc, ResourceEvent event)
> ResourceRequestEvent req = (ResourceRequestEvent) event
//将事件的类型投射恢复成 ResourceRequestEvent
> LocalizerContext ctxt = req.getContext() //从事件对象中恢复 LocalizerContext
> ContainerId container = ctxt.getContainerId() //从 LocalizerContext 中恢复容器
> rsrc.ref.add(container)
> e = new LocalizerResourceRequestEvent(rsrc, req.getVisibility(), ctxt,
req.getLocalResourceRequest().getPattern())
>> super(LocalizerEventType.REQUEST_RESOURCE_LOCALIZATION, ...)
//另创一个 REQUEST_RESOURCE_LOCALIZATION 事件
> rsrc.dispatcher.getEventHandler().handle(e) == ResourceLocalizationService.handle(e)
//用这个事件向 ResourceLocalizationService 提出请求
```

Hadoop 的这种风格, 逻辑上很是清晰, 但是效率上多少会受些影响。不过现在软件界总的倾向是宁可牺牲一些效率也要使代码易读易维护。

我们接着往下看 ResourceLocalizationService 对此事件的反应:

```
ResourceLocalizationService.handle(LocalizationEvent event)
> String locId = event.getLocalizerId() //每项具体的资源都有个 LocalizerId
> switch (event.getType()) {
> case REQUEST_RESOURCE_LOCALIZATION:
>+ // 0) find running localizer or start new thread
>+ req = (LocalizerResourceRequestEvent) event
>+ switch (req.getVisibility()) {
>+ case PUBLIC: //公共资源
>++ publicLocalizer.addResource(req) //公共资源的本地化交由 publicLocalizer 线程完成
>++> pending.put(queue.submit(new FSDownload(lfs, null, conf, publicDirDestPath,
resource, request.getContext().getStatCache()), request)
//创建一个 FSDownload 对象, 将其提交到队列 queue 中, 再放入 pending 队列
>+ case PRIVATE: case APPLICATION: //非公共资源
```

```

> ++ localizer = privLocalizers.get(locId) //获取非公共资源本地化线程
> ++ if (null == localizer) { //如果尚未创建则加以创建
> +++ LOG.info("Created localizer for " + locId)
> +++ localizer = new LocalizerRunner(req.getContext(), locId) //创建非公共资源本地化线程
> +++ privLocalizers.put(locId, localizer)
> +++ localizer.start() //启动这个线程
> ++ } //end if
> ++ // 1) propagate event
> ++ localizer.addResource(req) //交给非公共资源本地化线程去完成
> + } //end switch (req.getVisibility())
> } //end switch (event.getType())

```

公共资源的本地化是交由一个线程 `publicLocalizer` 完成的, 这个线程是一个定义于 `ResourceLocalizationService` 内部的 `PublicLocalizer` 类对象, 下面是这个类的摘要。不过这个线程本身并不直接从事资源的本地化, 而是维持着一个线程池, 每当有本地化资源请求到来时就为其创建一个可以作为线程运行的 `Callable` 对象 `FSDownload`, 负责具体的文件下载。

```

ResourceLocalizationService.PublicLocalizer extends Thread {}
] FileContext lfs
] Configuration conf
] ExecutorService threadPool //线程池
] CompletionService<Path> queue //等待被线程池中的线程加以执行的队列
] Map<Future<Path>, LocalizerResourceRequestEvent> pending
] PublicLocalizer(Configuration conf)
    > super("Public Localizer")
    > this.lfs = getLocalFileContext(conf)
    > this.conf = conf
    > this.pending = Collections.synchronizedMap(
        new HashMap<Future<Path>, LocalizerResourceRequestEvent>())
    > this.threadPool = createLocalizerExecutor(conf)
    > this.queue = new ExecutorCompletionService<Path>(threadPool)
] addResource(LocalizerResourceRequestEvent request)
    > LocalizedResource rsrc = request.getResource()
    > LocalResourceRequest key = rsrc.getRequest()
    > LOG.info("Downloading public rsrc:" + key)
    > % //下面的注解直接引自以下源代码
    / *
        * Here multiple containers may request the same resource. So we need
        * to start downloading only when
        * 1) ResourceState == DOWNLOADING
        * 2) We are able to acquire non blocking semaphore lock.
    */

```

```

    * If not we will skip this resource as either it is getting downloaded
    * or it FAILED / LOCALIZED.
    * /

> %
> if (rsrc.getState().equals(ResourceState.DOWNLOADING)) {
>+ LocalResource resource = request.getResource().getRequest()
>+ publicRootPath = dirsHandler.getLocalPathForWrite("." + Path.SEPARATOR +
    ContainerLocalizer.FILECACHE, ...) //“./filecache”
>+ publicDirDestPath = publicRsrc.getPathForLocalization(key, publicRootPath)
>+ if (!publicDirDestPath.getParent().equals(publicRootPath)) {
>++ DiskChecker.checkDir(new File(publicDirDestPath.toUri().getPath()))
>+ }
>+ d = new FSDownload(lfs, null, conf, publicDirDestPath,
    resource, request.getContext().getStatCache()), request)
    // FSDownload 是个 Callable, 是供线程调用的模块
    // 由线程池中的线程加以调用, 逻辑上可以视作线程
>+ f = queue.submit(d) //CompletionService.submit(), 作为 Future 任务提交, 等待被调用
>+ pending.put(f) // 放在待完成的 Future 任务集合中, 等待其完成并返回结果
> }
] run()
> while (!Thread.currentThread().isInterrupted()) {
>+ Future<Path> completed = queue.take() //等待 CompletionService 队列中有任务完成
>+ LocalizerResourceRequestEvent assoc = pending.remove(completed)
    //从 Future 集合 pending 中摘除, 并获取相关联的本地化请求
>+ Path local = completed.get()
>+ LocalResourceRequest key = assoc.getResource().getRequest()
>+ e = new ResourceLocalizedEvent(key, local, FileUtil.getDU(new File(local.toUri())))
>+> super(rsrc, ResourceEventType.LOCALIZED) //表示本项资源的本地化已经完成
>+ publicRsrc.handle(e) == LocalResourcesTrackerImpl.handle()
    //用 LOCALIZED 事件驱动 LocalResourcesTrackerImpl 的状态机
> } //end while

```

有公共资源需要本地化的时候, 就通过这里的 `addResource()` 将其挂入这个线程的 `queue` 队列。

该项公共资源实际的本地化是由 `FSDownload` 对象完成的。这里称之为 `Download` 倒是很贴切, 因为公共资源的源头都在 `RM` 节点上, 在 `NM` 节点上本地化一项公共资源就意味着从 `RM` 节点下载这项资源。`FSDownload` 是一个实现了 `Callable` 界面的类, 当被指派给线程池中的某个线程时, 就成为那个线程执行的程序。不同 `FSDownload` 对象的代码当然都一样, 只是创建对象时的参数不同。`Callable` 类的 `call()` 函数与线程的 `run()` 函数相似, 当其 `call()` 函数受到调用时就开始执行其代码, 所以我们着重看它的 `call()` 函数。

```

class FSDownload implements Callable<Path> {}
] FileContext files
] UserGroupInformation userUgi
] Configuration conf
] LocalResource resource //代表着需要加以本地化的资源,是作为参数传下来的
] LoadingCache<Path,Future<FileStatus>> statCache
] Path destDirPath //The local FS dir path under which this resource is to be localized to
//这个目标路径是在创建对象时作为参数传下来的
//资源的源路径则在 resource 中,可通过 resource.getResource()获取
] createDir(Path path, FsPermission perm)
] createStatusCacheLoader(final Configuration conf)
    > new CacheLoader<Path,Future<FileStatus>>()
        ] Future<FileStatus> load(Path path)
            > fs = path.getFileSystem(conf)
            > return Futures.immediateFuture(fs.getFileStatus(path))
] checkPublicPermsForAll(FileSystem fs, FileStatus status, FsAction dir, FsAction file)
] getFileStatus(final FileSystem fs, final Path path,
    LoadingCache<Path,Future<FileStatus>> statCache)
] copy(Path sCopy, Path dstdir) //复制文件,可以跨节点、跨文件系统复制
] unpack(File localrsrc, File dst)
] call()
    > sCopy = ConverterUtils.getPathFromYarnURL(resource.getResource())
    //获取该项公共资源的源路径
    //资源描述 resource 所提供的是一种 YarnURL,需要将其转换成 Hadoop 的 URI
    > createDir(destDirPath, cachePerms) //创建目标目录
    > dst_work = new Path(destDirPath + "_tmp") //加后缀_tmp 作为工作目录名
    > createDir(dst_work, cachePerms) //创建工作目录
    > dFinal = files.makeQualified(new Path(dst_work, sCopy.getName())) //目标文件名
    > dTmp = (null == userUgi) ?files.makeQualified(copy(sCopy, dst_work))
        : userUgi.doAs(new PrivilegedExceptionAction<Path>())
    ] run()
        > return files.makeQualified(copy(sCopy, dst_work))
    //实际的文件复制。如果有 userUgi,就通过 doAs()以具体用户的权限执行
    //不管是否通过 userUgi.doAs()执行,这里的核心就是 copy(),即文件的复制
    > unpack(new File(dTmp.toUri()), new File(dFinal.toUri()))
    > changePermissions(dFinal.getFileSystem(conf), dFinal)
    > files.rename(dst_work, destDirPath, Rename.OVERWRITE) //更改工作文件名
    > return files.makeQualified(new Path(destDirPath, sCopy.getName()))
    //返回本地的目标目录路径+文件名

```

这个 call() 函数的返回值,即最后形成的文件路径名,作为一个 Future 对象会留在

PublicLocalizer 的 CompletionService 队列 queue 中。

然后我们回到 PublicLocalizer.run() 的代码, 这个线程的每一轮循环都从 queue.take() 开始, 这个函数试图从该队列中拿到一个已经完成的 Future 任务, 如果没有就睡眠等待。这样, 当有 FSDownload.call() 返回的时候, 那个线程就被唤醒, 最后把所返回的路径名与相关的信息整合在一起, 产生一个类型为 LOCALIZED 的 ResourceLocalizedEvent 事件。在这个事件的触发下, LocalizedResource 会调用 FetchSuccessTransition.transition():

```
addTransition(ResourceState.DOWNLOADING, ResourceState.LOCALIZED,
    ResourceEventType.LOCALIZED, new FetchSuccessTransition())
```

显然, 这个事件意味着本项公共资源的本地化已经完成, 这时候才真正成为 Localized。

```
[LocalizedResource.FetchSuccessTransition.transition()]
```

```
FetchSuccessTransition.transition(LocalizedResource rsrc, ResourceEvent event)
> locEvent = (ResourceLocalizedEvent) event //这实际上是个 ResourceLocalizedEvent 对象
> rsrc.localPath = Path.getPathWithoutSchemeAndAuthority(locEvent.getLocation())
    //搭载在 ResourceLocalizedEvent 上的资源路径名是个带 SchemeAndAuthority 的 URI
    //现在从中抽取文件路径名的那一部分
> rsrc.size = locEvent.getSize()
> for (ContainerId container : rsrc.ref) {
>+ //给所涉及的每个容器发送一个 ContainerEventType.RESOURCE_LOCALIZED 事件
>+ e = new ContainerResourceLocalizedEvent(container, rsrc.rsrc, rsrc.localPath)
>+> super(container, ContainerEventType.RESOURCE_LOCALIZED, rsrc)
>+ rsrc.dispatcher.getEventHandler().handle(e) //用此事件驱动 ContainerImpl 的状态机
>+> ContainerImpl.LocalizedTransition.transition()
> }
```

一项资源往往为多个容器所共用, 从而出现在多个容器中, 所以这里要向每一个包含该项资源的容器都发送一个 RESOURCE_LOCALIZED 事件。而具体容器对此的反应, 则是执行 LocalizedTransition.transition(), 然后进入 LOCALIZED 状态 (如果其所含的所有资源都已完成本地化) 或停留在 LOCALIZING 状态 (如果容器中还有别的资源尚未完成本地化)。

```
addTransition(ContainerState.LOCALIZING,
    EnumSet.of(ContainerState.LOCALIZING, ContainerState.LOCALIZED),
    ContainerEventType.RESOURCE_LOCALIZED, new LocalizedTransition())
```

下面是 LocalizedTransition.transition() 的摘要:

```
LocalizedTransition.transition()
> ContainerResourceLocalizedEvent rsrcEvent = (ContainerResourceLocalizedEvent) event
> List<String> syms = container.pendingResources.remove(rsrcEvent.getResource())
    //将事件中所载已完成本地化的资源 (可有多项) 从 pendingResources 集合中摘除
> container.localizedResources.put(rsrcEvent.getLocation(), syms) //加入 localizedResources 中
```



```

> if (!container.pendingResources.isEmpty()) { //如果 pendingResources 集合非空,则
>+ return ContainerState.LOCALIZING           //同志仍须努力
> }
> container.sendLaunchEvent()                 //容器中所有资源均已本地化,可以投运了
> container.metrics.endInitingContainer()      //用于统计目的
> return ContainerState.LOCALIZED             //大功告成

```

摘要中加了注释,无须再多说了。

看完公共资源的本地化,再看非公共资源的本地化。

与公共资源不同的是,为每项非公共资源创建的是 LocalizerRunner 类的线程 localizer,下面是 LocalizerRunner 的摘要:

```

[ResourceLocalizationService.LocalizerTracker.handle() > new LocalizerRunner()]

ResourceLocalizationService.LocalizerRunner extends Thread {}
] LocalizerContext context
] String localizerId
] Map<LocalResourceRequest, LocalizerResourceRequestEvent> scheduled
] List<LocalizerResourceRequestEvent> pending
] ContainerExecutor exec
] addResource(LocalizerResourceRequestEvent request)
  > pending.add(request)
] update(List<LocalResourceStatus> remoteResourceStatuses)
] run()
  > nmPrivateCTokensPath = dirsHandler.getLocalPathForWrite(...)
  > //1) write credentials to private dir
  > writeCredentials(nmPrivateCTokensPath)
  > // 2) exec initApplication and wait
  > List<String> localDirs = getInitializedLocalDirs()
  > if (dirsHandler.areDisksHealthy()) {
  >+ exec.startLocalizer(nmPrivateCTokensPath, localizationServerAddress, ...)
  >+ // exec 是一个 ContainerExecutor,这是抽象类型,其实际类型取决于配置文件
  > }
  > delService.delete(null, nmPrivateCTokensPath, new Path[] { })
] ...

```

我们在前面 ResourceLocalizationService.LocalizerTracker.handle()中看到,对于非公共资源,在创建了 LocalizerRunner 线程之后就调用 localizer.start(),这其实是 Thread.start(),这样就导致了 LocalizerRunner.run()的执行。而这个 run()函数的核心,则在于对 exec.startLocalizer()的调用。

这里的 exec 是 ResourceLocalizationService 内部的一个 ContainerExecutor 对象,但是

ContainerExecutor 只是一个抽象类型, YARN 在此基础上提供了三种实际落地的类型, 即 DefaultContainerExecutor、DockerContainerExecutor 和 LinuxContainerExecutor, 可以在配置文件 yarn-default.xml 中的“yarn.nodemanager.container-executor.class”条目中加以指定, 一般采用 DefaultContainerExecutor。我们假定采用 DefaultContainerExecutor:

```
[LocalizerRunner.run() > DefaultContainerExecutor.startLocalizer]
```

```
void startLocalizer(Path nmPrivateContainerTokensPath,
    InetAddress nmAddr, String user, String appId, String locId,
    LocalDirsHandlerService dirsHandler) throws IOException, InterruptedException {
    List<String> localDirs = dirsHandler.getLocalDirs(); // 目标目录路径的集合
    List<String> logDirs = dirsHandler.getLogDirs(); // 用于 LOG 的目录路径集合
    createUserLocalDirs(localDirs, user); // 在本地创建用户的目标文件目录
    createUserCacheDirs(localDirs, user); // 创建该用户的缓存目录
    createAppDirs(localDirs, user, appId); // 创建针对具体 AppId 的文件目录
    createAppLogDirs(appId, logDirs, user); // 创建该 App 的日志文件目录

    // randomly choose the local directory
    Path appStorageDir = getWorkingDir(localDirs, user, appId); // 创建一个临时的工作目录

    // 处理与身份证件 Token 有关的文件
    String tokenFn = String.format(ContainerLocalizer.TOKEN_FILE_NAME_FMT, locId);
    Path tokenDst = new Path(appStorageDir, tokenFn);
    copyFile(nmPrivateContainerTokensPath, tokenDst, user);
    LOG.info("Copying from " + nmPrivateContainerTokensPath + " to " + tokenDst);

    // 进行本地化
    FileContext localizerFc = FileContext.getFileContext(
        lfs.getDefaultFileSystem(), getConf());
    localizerFc.setUMask(lfs.getUMask());
    localizerFc.setWorkingDirectory(appStorageDir); // 转入临时工作目录
    LOG.info("Localizer CWD set to " + appStorageDir + " =
        " + localizerFc.getWorkingDirectory())
    ContainerLocalizer localizer = new ContainerLocalizer(localizerFc, user, appId, locId,
        getPaths(localDirs), RecordFactoryProvider.getRecordFactory(getConf()));
    // 创建一个 ContainerLocalizer 类对象
    // TODO: DO it over RPC for maintaining similarity?
    localizer.runLocalization(nmAddr); // 调用 ContainerLocalizer.runLocalization()
}
```

这里先为本地化做一些必要的准备, 然后创建一个 ContainerLocalizer 类对象 localizer 加

以具体实施。ContainerLocalizer 带有 main() 函数, 可以被独立作为 JVM 进程运行, 但是这里通过 new 直接创建就绕过了它的 main() 函数, 而只是将其作为一个模块使用。此时, 其所提供的方法函数 runLocalization() 是在 LocalizerRunner 这个线程的上下文中执行的:

```
class ContainerLocalizer {}

] String user           //容器所属的用户
] String appId          //容器所属的 App
] List<Path> localDirs   //有关的本地目录
] String localizerId
] FileContext lfs
] RecordFactory recordFactory
] Map<LocalResource,Future<Path>> pendingResources //有待本地化的资源
] String appCacheDirContextName
] getProxy(final InetAddress nmAddr) //资源本地化常常涉及跨节点操作
    > YarnRPC rpc = YarnRPC.create(conf)
    > return rpc.getProxy(LocalizationProtocol.class, nmAddr, conf)
] runLocalization(final InetAddress nmAddr) //这是主要的调用入口
    > initDirs(conf, user, appId, lfs, localDirs)
    > Path tokenPath = new Path(String.format(TOKEN_FILE_NAME_FMT, localizerId))
    > credFile = lfs.open(tokenPath)
    > creds.readTokenStorageStream(credFile)
    > UserGroupInformation remoteUser = UserGroupInformation.createRemoteUser(user)
    > remoteUser.addToken(creds.getToken(LocalizerTokenIdentifier.KIND))
    > LocalizationProtocol nodeManager = //获取通向对方 NodeManager 的 Proxy
        remoteUser.doAs(new PrivilegedAction<LocalizationProtocol>()
    ] run()
        > return getProxy(nmAddr) //返回 LocalizationProtocolPBClientImpl
    > ugi = UserGroupInformation.createRemoteUser(user)
    > for (Token<?extends TokenIdentifier> token : creds.getAllTokens()) ugi.addToken(token)
    > ExecutorService exec = createDownloadThreadPool() //创建 ExecutorService 线程池
    >> t = new ThreadFactoryBuilder().setNameFormat("ContainerLocalizer Downloader").build()
    >> return Executors.newSingleThreadExecutor(t)
    > CompletionService<Path> ecs = createCompletionService(exec)
        //在线程池基础上创建 CompletionService, 用来接受和运行 callable
    >> return new ExecutorCompletionService<Path>(exec)
    > localizeFiles(nodeManager, ecs, ugi)
] localizeFiles(LocalizationProtocol nodemanager, CompletionService<Path> cs,
                                                         UserGroupInformation ugi)

    > while (true) {
    >+ LocalizerStatus status = createStatus() //逐项列出对彼岸节点的资源要求
    >+ response = nodemanager.heartbeat(status) //通过 PB 和 RPC 机制向对方发送心跳
```

```

>+ switch (response.getLocalizerAction()) { //根据对方回应分情形处理
>+ case LIVE: //对方还活着
>+ List<ResourceLocalizationSpec> newRsrcs = response.getResourceSpecs()
>+ for (ResourceLocalizationSpec newRsrc : newRsrcs) { //对于彼岸发回的每项资源描述
>+ if (!pendingResources.containsKey(newRsrc.getResource())) {
>+ d = download(new Path(newRsrc.getDestinationDirectory().getFile()),
newRsrc.getResource(), ugi) //创建 Callable 对象 FSDownload
>+ > DiskChecker.checkDir(new File(path.toUri().getRawPath()))
>+ > new FSDownload(lfs, ugi, conf, path, rsrc)
>+ f = cs.submit(d) //将其提交给 CompletionService,使其作为 Future 任务执行
>+ pendingResources.put(newRsrc.getResource(), f) //放在 pendingResources 集合中
>+ }
>+ } //end for
>+ break
>+ case DIE: //对方已死
>+ // killall running localizations
>+ for (Future<Path> pending : pendingResources.values()) {
>+ pending.cancel(true)
>+ }
>+ status = createStatus()
>+ nodemanager.heartbeat(status)
>+ return
>+ }
>+ cs.poll(1000, TimeUnit.MILLISECONDS) //睡眠直至 cs 中有 Future 任务完成或超时
> } //end while(true)
] main(String[] argv)
> localizer = new ContainerLocalizer(FileContext.getLocalFSFileContext(),
user, appId, locId, localDirs, ...)
> localizer.runLocalization(nmAddr)

```

先看 runLocalization()的摘要。撇开与身份和安全有关的措施不说,我们在这里关心的就是如何把非公共资源作为文件从另一个节点拷贝到本地的问题。公共资源的源头一定在 RM 节点上,所以前面我们看到的本地化都是从 RM 节点拷贝文件。这比较简单,因为 NM 节点与 RM 节点之间本来就有通信。但是非公共资源的源头却有可能在集群内的任何一个节点上,这就要复杂一些了,因为不同 NM 节点之间平时是没有通信的。正因为这样,我们需要与资源所在的节点临时建立通信管道。这里的 nodeManager 是一个实现了 LocalizationProtocol 界面的对象,实际上是 LocalizationProtocolPBClientImpl。显然,这实际上是个采用 ProtoBuf 的 RPC 通道,但是它代表着对方,所以名曰 nodeManager,实际上是对方在本节点上的代理。建立好 RPC 通道之后,接着就创建一个为下载服务的线程池,这是一个实现了 ExecutorService 界面的对象,并在这个池中创建一个线程。然后以此为基础创建一个

个实现了 CompletionService 界面的 ExecutorCompletionService 对象。这都是在为接受并执行 Callable 任务模块做准备,有关的这些机制都是由 JDK 提供的。做好这些准备之后,下面就是 localizeFiles()的事了。

如代码中所示,localizeFiles()是由 ContainerLocalizer 提供的一个方法(函数),后者则是由 ContainerExecutor 创建,为 ContainerExecutor 所调用的。而 ContainerExecutor 所提供的那些函数,则都是在线程 ResourceLocalizationService.LocalizerRunner 的上下文中被调用。另一方面,localizeFiles()的主体是个无限循环,这实质上就是 LocalizerRunner 的主循环。

每一轮循环都以发送心跳报告开始,不过这不是向 RM 节点发送,而是向资源所在的 NM 节点发送,所得的回应可以让我们知道对方是否还活着。如果还活着的话,就通过 download()为需要从该结点下载的每项资源创建一个 Callable 对象 FSDownload,并将其提交给前面已经准备好了的 CompletionService,成为一项 Future 任务,同时又将这个 Future 任务放进 pendingResources 集合。所谓 pendingResources,就是尚未(从该目标结点)完成本地化的资源。

可见,非公共资源的本地化操作最终也是由 FSDownload 完成的,这跟公共资源的本地化操作一样。不同的只是,公共资源一定在 RM 节点上,NM 节点与 RM 节点本来就有联系,而非公共资源有可能在别的 NM 节点上。

注意,FSDownload 只是针对单项资源的,而实际上需要本地化的资源可能有好多,所以每一项资源都会有一个独立的线程在执行着 FSDownload.call(),各自负责一项具体资源的下载,并发地处理着多项资源的本地化。

这里还要补叙一下 LocalizerRunner.update()。由 ContainerLocalizer.localizeFiles()在此岸通过 RPC 发出的心跳报告,到了彼岸就是由 LocalizerRunner.update()接应和处理的。

```
LocalizerRunner.update(List<LocalResourceStatus> remoteResourceStatuses)
> response = recordFactory.newRecordInstance(LocalizerHeartbeatResponse.class)
//生成一个有待填写具体内容的心跳响应
> user = context.getUser()
> applicationId = context.getContainerId().getApplicationAttemptId().getApplicationId()
> if (remoteResourceStatuses.isEmpty()) {
>+ LocalResource next = findNextResource()
>+ if (next != null) {
>++ response.setLocalizerAction(LocalizerAction.LIVE)
>++ rsrcs = new ArrayList<ResourceLocalizationSpec>()
>++ rsrc = NodeManagerBuilderUtils.newResourceLocalizationSpec(
next, getPathForLocalization(next))
>++ rsrcs.add(rsrc)
>++ response.setResourceSpecs(rsrcs)
>+ } else if (pending.isEmpty()) {
>++ response.setLocalizerAction(LocalizerAction.DIE)
>+ } else {
```

```

> ++ response.setLocalizerAction(LocalizerAction.LIVE)
> + }
> + return response
> }

> rsrcs = new ArrayList<ResourceLocalizationSpec>() //资源描述的集合
> for (LocalResourceStatus stat: remoteResourceStatuses) { //对于对方所要求的每项资源
> + LocalResource rsrc = stat.getResource()
> + req = new LocalResourceRequest(rsrc)
> + LocalizerResourceRequestEvent assoc = scheduled.get(req)
> + if (assoc == null) {
> ++ LOG.error("Unknown resource reported: " + req)
> ++ continue
> + }
> + switch (stat.getStatus()) {
> + case FETCH_SUCCESS:
> ++ e = new ResourceLocalizedEvent(req,
        ConverterUtils.getPathFromYarnURL(stat.getLocalPath()), stat.getLocalSize())
> ++ getLocalResourcesTracker(req.getVisibility(), user, applicationId).handle(e)
        == LocalResourcesTrackerImpl.handle(e)
> ++ scheduled.remove(req)
> ++ if (pending.isEmpty()) {
> +++ response.setLocalizerAction(LocalizerAction.DIE)
> +++ break
> +++ }
> ++ response.setLocalizerAction(LocalizerAction.LIVE)
> ++ LocalResource next = findNextResource()
> ++ if (next != null) { //资源存在,创建对于本项资源的描述
> +++ ResourceLocalizationSpec resource =
        NodeManagerBuilderUtils.newResourceLocalizationSpec(
            next, getPathForLocalization(next))
> +++ rsrcs.add(resource) //将资源描述加入 rsrcs 集合
> ++ }
> + case FETCH_PENDING:
> ++ response.setLocalizerAction(LocalizerAction.LIVE)
> + case FETCH_FAILURE:
> ++ response.setLocalizerAction(LocalizerAction.DIE)
> ++ ...
> + } //end switch
> } //end for (LocalResourceStatus stat : remoteResourceStatuses)
> response.setResourceSpecs(rsrcs) //把这些资源描述搭载在心跳响应上

```



```
> return response
```

这时候的心跳报告像个询问单,里面逐项列出了对本结点的资源要求。这里就在逐项考察这些资源是否存在,如果存在就为其创建一个资源描述,即 ResourceLocalizationSpec。最后把这些资源描述搭载在心跳响应上发回给对方,让对方前来下载。

这样,通过节点间的交互,在完成了所有资源的本地化之后,每一项资源都已发生一次 FetchSuccessTransition 跳变,从而使本地凡是需要用到此项资源的容器都得到一次 RESOURCE_LOCALIZED 事件的触发。而每个容器的状态机对于这个事件的反应规则则如前所述是调用 LocalizedTransition.transition(),然后根据容器中的所有资源是否都已完成本地化而或转入 LOCALIZED 状态,或者停留在 LOCALIZING 状态。换言之,凡是处于 LOCALIZING 状态的容器,每受到一次 ContainerEventType.RESOURCE_LOCALIZED 事件的触发,即每有一项资源完成本地化,就会调用一次 LocalizedTransition.transition(),但是其后续状态有可能停留在 LOCALIZING 不变,也可能变成 LOCALIZED。这是因为一个容器通常都会有多项需要本地化的资源,只有在所有资源都已本地化时才能转入 LOCALIZED 状态。

到了本容器的 pendingResources 集合为空的时候,这个容器的所有资源都已经到位,整个容器的本地化就完成了。于是就会通过 sendLaunchEvent() 向这个容器发送一个 LAUNCH_CONTAINER 事件,下一步是发起该容器运行的事了。

```
[ContainerImpl.LocalizedTransition.transition() > ContainerImpl.sendLaunchEvent()]
```

```
void sendLaunchEvent() {
    ContainersLauncherEventType launcherEvent =
        ContainersLauncherEventType.LAUNCH_CONTAINER;
    if (recoveredStatus == RecoveredContainerStatus.LAUNCHED) {
        // try to recover a container that was previously launched
        launcherEvent = ContainersLauncherEventType.RECOVER_CONTAINER;
    }
    dispatcher.getEventHandler().handle(new ContainersLauncherEvent(this, launcherEvent));
}
```

一般,对于新指派的容器,这里创建的事件是 LAUNCH_CONTAINER,实际上是发起容器运行的请求。但是这里还有一种特殊情况,就是这个容器以前就已经被发起运行,但是因为某种原因而被再次发起,那就是恢复其运行的事了,所以在这种情况下要把事件改成 RECOVER_CONTAINER。这个事件的发送目标是 ContainersLauncher。

ContainerManagerImpl 在其构造函数中创建了一个 ContainersLauncher 对象 containersLauncher,并登记为 ContainersLauncherEventType 的事件处理器(EventHandler)。而 ContainerImpl 对象也是由 ContainerManagerImpl 创建的,二者使用同一个 dispatcher。所以这里代码中的 dispatcher.getEventHandler() 就是 containersLauncher。ContainersLauncher 类的摘要如下:

```
[ContainerImpl.LocalizedTransition.transition() > ContainerImpl.sendLaunchEvent()]
```

```

=> ContainersLauncher.handle()]

class ContainersLauncher extends AbstractService implements EventHandler<...>{}
] ContainerExecutor exec
] Dispatcher dispatcher
] ExecutorService containerLauncher = Executors.newCachedThreadPool(
    new ThreadFactoryBuilder().setNameFormat("ContainersLauncher # %d").build())
] Map<ContainerId, ContainerLaunch> running =
    Collections.synchronizedMap(new HashMap<ContainerId, ContainerLaunch>())
] serviceInit(Configuration conf)
] handle(ContainersLauncherEvent event)
    > Container container = event.getContainer()
    > ContainerId containerId = container.getContainerId()
    > switch (event.getType()) {
    > case LAUNCH_CONTAINER:
    >+ Application app = context.getApplications().get(
        containerId.getApplicationAttemptId().getApplicationId())
    >+ ContainerLaunch launch = new ContainerLaunch(context, getConfig(), dispatcher,
        exec, app, event.getContainer(), dirsHandler, containerManager);
    >+ containerLauncher.submit(launch);
        //将 callable 对象 ContainerLaunch 提交给 ExecutorService
    >+ running.put(containerId, launch)    //并将其记录在 running 集合中
    >+ break
    > case RECOVER_CONTAINER
    >+ ...
    > case CLEANUP_CONTAINER;
    >+ ...
    > }

```

我们在这里只关心 ContainersLauncher 对象对 LAUNCH_CONTAINER 事件的反应。显然,这里的核心在于创建一个实现了 Callable 界面的 ContainerLaunch 类对象,并将其提交给线程池 containerLauncher。这样,线程池中的某个线程就会调用 ContainerLaunch.call(), 那以后就是 NM 节点上容器投运的事了。

在深入到 ContainerLaunch 中之前,我们还要回顾一下 ApplicationImpl 状态机的变化。前面 ContainerManagerImpl.startContainers()的代码中调用了 startContainerInternal()。后者显然是前者 startContainers()的实现细节。而在 startContainerInternal()的代码中,在向具体 ApplicationImpl 的状态机发送 INIT_APPLICATION 事件之后又向同一状态机发送了一个 INIT_CONTAINER 事件,二者同属 ApplicationEventType。前者使 ApplicationImpl 的状态机执行 AppInitTransition.transition(),并进入 INITING 状态,后来又进入 RUNNING 状

态。而 `AppInitTransition.transition()` 的执行,则引出了围绕着资源本地化的一系列事件和操作。那么 `INIT_CONTAINER` 事件又会引出一些什么样的事件和操作呢? 我们可以看到,在 `ApplicationImpl` 的状态机中有这么几条规则:

```
addTransition(ApplicationState.NEW, ApplicationState.NEW,
    ApplicationEventType.INIT_CONTAINER,
    new InitContainerTransition())

addTransition(ApplicationState.INITING, ApplicationState.INITING,
    ApplicationEventType.INIT_CONTAINER,
    new InitContainerTransition())

addTransition(ApplicationState.RUNNING, ApplicationState.RUNNING,
    ApplicationEventType.INIT_CONTAINER,
    new InitContainerTransition())
```

这就是说,不管 `ApplicationImpl` 是处于 `NEW`、`INITING` 还是 `RUNNING` 状态,只要有 `INIT_CONTAINER` 事件到来,就总要执行一下 `InitContainerTransition.transition()`,但是其状态却维持不变。

```
[ContainerManagerImpl.startContainers() > startContainerInternal()
=> ApplicationImpl.InitContainerTransition.transition()]
```

```
InitContainerTransition.transition(ApplicationImpl app, ApplicationEvent event)
> initEvent = (ApplicationContainerInitEvent) event
> Container container = initEvent.getContainer()
> app.containers.put(container.getContainerId(), container)
> LOG.info("Adding " + container.getContainerId() + " to application " + app.toString())
> switch (app.getApplicationState()) {
> case RUNNING:
>+ e = new ContainerInitEvent(container.getContainerId())
>+> super(c, ContainerEventType.INIT_CONTAINER) //注意,这是 ContainerEventType
>+ app.dispatcher.getEventHandler().handle(e) //这里所触发的是 ContainerImpl 的状态机
>+> ContainerImpl.RequestResourcesTransition.transition()
> case INITING: case NEW:
>+ break //无操作
> }
```

搭载在事件 `event` 上的是一个 `Container` 对象,更确切地说是实现了 `Container` 界面的 `ContainerImpl` 对象。这里首先要把它加入 `ApplicationImpl` 的 `containers` 集合中去,这个 `containers` 是一个 `Map`,里面的每个元素都是一对 `ContainerId` 和 `Container`,把相同的元素重复 `put()` 几次都没有关系。

然后就要看这个 ApplicationImpl 处于什么状态了,只有在 RUNNING 状态下才有进一步的操作,那就是向这个 ContainerImpl 发送一个 INIT_CONTAINER 事件。注意,这次发送的是 ContainerEventType.INIT_CONTAINER,而不是 ApplicationEventType.INIT_CONTAINER 了。

7.4 容器的投运

现在要深入 ContainerLaunch 这个类的内部去考察容器投运了。下面是这个类的摘要。

```
class ContainerLaunch implements Callable<Integer> {}
] String CONTAINER_SCRIPT = Shell.appendScriptExtension("launch_container")
] String FINAL_CONTAINER_TOKENS_FILE = "container_tokens"
] String PID_FILE_NAME_FMT = "% s.pid"
] String EXIT_CODE_FILE_SUFFIX = ".exitcode"
] ContainerExecutor exec
] Application app
] Container container
] ContainerManagerImpl containerManager
] call()          //作为一个 Callable,这是受调用执行程序的总入口
] abstract class ShellScriptBuilder {}
]] create()
    > return Shell.WINDOWS ?new WindowsShellScriptBuilder()
                        : new UnixShellScriptBuilder()
] class UnixShellScriptBuilder extends ShellScriptBuilder {}
]] UnixShellScriptBuilder()
    > line("#!/bin/bash")
    > line()
]] command(List<String> command)
    > line("exec /bin/bash -c \"", StringUtils.join(" ", command), "\"")
    > errorCheck()
]] env(String key, String value)
    > line("export ", key, " = \"", value, "\"")
]] ...
```

前面 ContainersLauncher.handle() 把一个 ContainerLaunch 对象通过 submit() 提交给 ExecutorService。将一个 Callable 提交(submit)给一个线程池之后,一旦线程池中有空闲的线程,就会让那个线程调用这个 Callable 的 call() 函数,所以 ContainerLaunch.call() 早晚会受到调用。我们知道,call() 在那个线程中的作用和地位类似于 run()。下面我们就来看这个函数的代码,这是一大段程序:

```
[ContainerImpl.LocalizedTransition.transition() > ContainerImpl.sendLaunchEvent()
=> ContainersLauncher.handle() > ContainerLauncher.submit() => ContainerLaunch.call()]
```

```

Integer ContainerLaunch.call() {
    final ContainerLaunchContext launchContext = container.getLaunchContext();
    //从容器中获取 CLC
    Map<Path,List<String>> localResources = null;
    ContainerId containerID = container.getContainerId();
    String containerIdStr = ConverterUtils.toString(containerID);
    final List<String> command = launchContext.getCommands(); //从 CLC 中获取命令行
    int ret = -1;

    // CONTAINER_KILLED_ON_REQUEST should not be missed if the container
    // is already at KILLING
    if (container.getContainerState() == ContainerState.KILLING) { //不幸,尚未投运就被 kill
        dispatcher.getEventHandler().handle(new ContainerExitEvent(containerID,
            ContainerEventType.CONTAINER_KILLED_ON_REQUEST,
            Shell.WINDOWS?ExitCode.FORCE_KILLED.getExitCode() :
                ExitCode.TERMINATED.getExitCode(),
            "Container terminated before launch."));
        return 0;
    }

    try {
        localResources = container.getLocalizedResources();
        if (localResources == null) { //资源本地化尚未成功
            throw RPCUtil.getRemoteException(
                "Unable to get local resources when Container " + containerID +
                " is at " + container.getContainerState());
        }

        final String user = container.getUser();
        //Variable expansion before the container script gets written out.
        List<String> newCmds = new ArrayList<String>(command.size());
        //创建新命令行,因为需要根据环境修正命令行中的一些元素
        String appIdStr = app.getAppId().toString();
        String relativeContainerLogDir = ContainerLaunch.getRelativeContainerLogDir(
            appIdStr, containerIdStr);
        Path containerLogDir = dirsHandler.getLogPathForWrite(relativeContainerLogDir, false);
        for (String str : command) {
            //把老命令行中的元素逐一根据环境需要加以修正后转移到新命令行
            // TODO: Should we instead work via symlinks without this grammar?
            newCmds.add(expandEnvironment(str, containerLogDir)); //加入新命令行
        }
    }
}

```

```

    }
    launchContext.setCommands(newCmds); //把新命令行写回 CLC

    Map<String, String> environment = launchContext.getEnvironment();
    // Make a copy of env to iterate & do variable expansion
    for (Entry<String, String> entry : environment.entrySet()) { //逐一修正环境变量
        String value = entry.getValue();
        value = expandEnvironment(value, containerLogDir);
        entry.setValue(value);
    }
    //End of variable expansion

    FileContext lfs = FileContext.getLocalFSFileContext(); //决定采用何种文件系统

    Path nmPrivateContainerScriptPath = dirsHandler.getLocalPathForWrite(
        getContainerPrivateDir(appIdStr, containerIdStr) + Path.SEPARATOR
        + CONTAINER_SCRIPT); //启动脚本路径,如“xyz/launch_container”
    Path nmPrivateTokensPath = dirsHandler.getLocalPathForWrite(
        getContainerPrivateDir(appIdStr, containerIdStr) + Path.SEPARATOR
        + String.format(ContainerLocalizer.TOKEN_FILE_NAME_FMT,
            containerIdStr)); //Token 文件路径
    Path nmPrivateClasspathJarDir = dirsHandler.getLocalPathForWrite(
        getContainerPrivateDir(appIdStr, containerIdStr)); //JAR 文件目录路径
    DataOutputStream containerScriptOutputStream = null;
    DataOutputStream tokensOutputStream = null;

    // Select the working directory for the container
    Path containerWorkDir =
        dirsHandler.getLocalPathForWrite(ContainerLocalizer.USERCACHE
            + Path.SEPARATOR + user + Path.SEPARATOR
            + ContainerLocalizer.APPCACHE + Path.SEPARATOR + appIdStr
            + Path.SEPARATOR + containerIdStr,
            LocalDirAllocator.SIZE_UNKNOWN, false); //工作目录路径

    String pidFileSubpath = getPidFileSubpath(appIdStr, containerIdStr); //PID 文件子路径

    // pid file should be in nm private dir so that it is not accessible by users
    pidFilePath = dirsHandler.getLocalPathForWrite(pidFileSubpath); //PID 文件全路径
    List<String> localDirs = dirsHandler.getLocalDirs(); //本地目录
    List<String> logDirs = dirsHandler.getLogDirs(); //日志目录

```



```

List<String> containerLogDirs = new ArrayList<String>();
for( String logDir : logDirs) {
    containerLogDirs.add(logDir + Path.SEPARATOR + relativeContainerLogDir);
}

if (!dirsHandler.areDisksHealthy()) {    //检查硬盘是否正常
    ret = ContainerExitStatus.DISKS_FAILED;
    throw new IOException("Most of the disks failed. "
        + dirsHandler.getDisksHealthReport(false));    //磁盘异常
}

try {
    //Write out the container - script in the nmPrivate space.
    List<Path> appDirs = new ArrayList<Path>(localDirs.size());
    for (String localDir : localDirs) {
        Path usersdir = new Path(localDir, ContainerLocalizer.USERCACHE);
        Path userdir = new Path(usersdir, user);
        Path appsdire = new Path(userdir, ContainerLocalizer.APPCACHE);
        appDirs.add(new Path(appsdire, appIdStr));
    }
    containerScriptOutputStream = lfs.create(
        nmPrivateContainerScriptPath, EnumSet.of(CREATE, OVERWRITE));
        //相当于创建并打开文件,用于写出本容器的启动脚本

    // Set the token location too.
    environment.put(
        ApplicationConstants.CONTAINER_TOKEN_FILE_ENV_NAME,
        new Path(containerWorkDir,
            FINAL_CONTAINER_TOKENS_FILE).toUri().getPath());
        //设置环境变量 HADOOP_TOKEN_FILE_LOCATION
        //变量值为工作目录下的“container_tokens”
    // Sanitize the container's environment
    sanitizeEnv(environment, containerWorkDir, appDirs, containerLogDirs,
        localResources, nmPrivateClasspathJarDir);
        //设置一系列的环境变量

    // Write out the environment
    exec.writeLaunchEnv(containerScriptOutputStream, environment, localResources,
        launchContext.getCommands());
        //把环境变量的设置,以及 launchContext 中带来的命令行写入脚本

```

```

//End of writing out container - script

//Write out the container - tokens in the nmPrivate space
tokensOutputStream = lfs.create(nmPrivateTokensPath, EnumSet.of(CREATE, OVERWRITE));
Credentials creds = container.getCredentials();
creds.writeTokenStorageToStream(tokensOutputStream); //有关安全机制
//End of writing out container - tokens
} finally {
    IOUtils.cleanup(LOG, containerScriptOutputStream, tokensOutputStream);
}

// LaunchContainer is a blocking call. We are here almost means the
// container is launched, so send out the event. 向 ContainerImpl 发送事件
dispatcher.getEventHandler().handle(new ContainerEvent(
    containerID, ContainerEventType.CONTAINER_LAUNCHED));
    //使 ContainerImpl 对象的状态机执行 LaunchTransition.transition(),
    //并进入 RUNNING 状态
context.getNMStateStore().storeContainerLaunched(containerID);

// Check if the container is signalled to be killed.
if (!shouldLaunchContainer.compareAndSet(false, true)) { //万一容器已被 Kill
    LOG.info("Container " + containerIdStr + " not launched as "
        + "cleanup already called");
    ret = ExitCode.TERMINATED.getExitCode();
}
else { //正常情况,发起容器运行
    exec.activateContainer(containerID, pidFilePath); //允许发起 Container 运行
    ret = exec.launchContainer(container, nmPrivateContainerScriptPath,
        nmPrivateTokensPath, user, appIdStr, containerWorkDir, localDirs, logDirs);
        //发起运行,这是个“blocking call”,当前线程将被阻塞
        //直至运行结束才会从这个函数调用返回
}
} catch (Throwable e) {
    LOG.warn("Failed to launch container.", e);
    dispatcher.getEventHandler().handle(new ContainerExitEvent(
        containerID, ContainerEventType.CONTAINER_EXITED_WITH_FAILURE, ret,
        e.getMessage()));
    return ret;
} finally { //运行结束之后的善后操作
    completed.set(true); //表示运行已结束
}

```

```

exec.deactivateContainer(containerID);           // 禁止再次发起 Container 运行
try {
    context.getNMStateStore().storeContainerCompleted(containerID, ret);
    // 存储结束时的状态
} catch (IOException e) {
    LOG.error("Unable to set exit code for container " + containerID);
}
}

if (LOG.isDebugEnabled()) {
    LOG.debug("Container " + containerIdStr + " completed with exit code " + ret);
}

if (ret == ExitCode.FORCE_KILLED.getExitCode()
    || ret == ExitCode.TERMINATED.getExitCode()) { // 因容器被 Kill 而失败
    // If the process was killed, Send container_cleanedup_after_kill and
    // just break out of this method.
    dispatcher.getEventHandler().handle(
        new ContainerExitEvent(containerID,
            ContainerEventType.CONTAINER_KILLED_ON_REQUEST, ret,
            "Container exited with a non - zero exit code " + ret));
    return ret;
}

if (ret != 0) { // 运行失败
    LOG.warn("Container exited with a non - zero exit code " + ret);
    this.dispatcher.getEventHandler().handle(new ContainerExitEvent(
        containerID,
        ContainerEventType.CONTAINER_EXITED_WITH_FAILURE, ret,
        "Container exited with a non - zero exit code " + ret));
    return ret;
}

LOG.info("Container " + containerIdStr + " succeeded "); // 运行成功
dispatcher.getEventHandler().handle(new ContainerEvent(containerID,
    ContainerEventType.CONTAINER_EXITED_WITH_SUCCESS));
// 使 ContainerImpl 对象的状态机执行 ExitedWithSuccessTransition.transition(),
// 并进入 CONTAINER_EXITED_WITH_SUCCESS 状态
return 0;
}

```

程序有点长,要分几个部分来看。

第一部分是从事件对象(重温一下,对象就是数据结构加操作函数)中抽取其 ContainerLaunchContext 即 CLC,再从 CLC 中抽取命令行,并进行一些条件的检验,包括容器是否已被 kill,以及容器的资源本地化是否已经完成。

第二部分是通过 expandEnvironment()对命令行和环境变量进行一些修正,生成一个新命令行并将其写回 CLC。当然,这个修正不会改变命令行的实质内容。

第三部分则是一些目录或文件路径的生成。其一为 nmPrivateContainerScriptPath,这是有待生成的容器脚本文件的路径,前缀 nm 显然是指 NodeManager。容器脚本文件名固定为“launch_container”,所在目录则需要根据 appId 和 containerId 加以生成。其二为供安全机制使用的 Token 文件的路径,Token 文件都有扩充名.tokens。其三是 nmPrivateClasspathJarDir,这显然就是供 JVM 执行的 JAR 文件所在目录。下面还要生成一批目标端的路径,包括 containerWorkDir、pidFilePath、localDirs、logDirs,还有 containerLogDirs、appDirs。其中 appDirs 包括 usersdir、userdir、appsdir。注意,此刻这些路径都还只是字符串而并未实际创建。接着就以 nmPrivateContainerScriptPath 为路径创建了一个输出流 containerScriptOutputStream,这意味着创建文件并按写模式打开,这是准备用来写启动脚本的。

第四部分接着就通过这个输出流将环境变量、本地化以后的资源路径以及来自 CLC 的命令行写入脚本。这样,就有了一个为投运本容器所需的启动脚本,供 NM 用来在本地启动一个命令行。此外,也以 nmPrivateTokensPath 为路径创建一个输出流 tokensOutputStream,并将有关的信息写入该文件。

第五部分是实质性的操作,先向 ContainerImpl 发送 CONTAINER_LAUNCHED 事件,然后就调用 exec.launchContainer(),后者的作用就是在宿主操作系统上发起执行一个命令行。按理说这个事件应该在调用了 exec.launchContainer()之后才能发出,但是正如注解中所说,对 launchContainer()的调用是个阻塞的“blocking call”,要到所启动的命令行执行完毕而退出时才能返回,所以就只好提前先发送事件了。

那以后的代码都是 exec.launchContainer()返回之后的善后操作,那时候这个容器已经被发起了。所谓发起一个容器,是指在宿主操作系统上发起一个 Shell 进程来执行以容器(实际上在这里是 CLC)中所带的命令行为核心的启动脚本,在这里是启动一个执行 AM 的 JVM 进程。至于容器中所带的其他资源,则是为该命令行所用。

我们在上面看到,ContainerLaunchContext 带来的只是一个命令行,而启动脚本是在本地通过 writeLaunchEnv()生成并写入脚本文件的。我们看一下这个函数的摘要:

```
[ContainerLaunch.call() > ContainerExecutor.writeLaunchEnv()]
```

```
ContainerExecutor.writeLaunchEnv(OutputStream out, Map<String, String> environment,
    Map<Path, List<String>> resources, List<String> command)
> ContainerLaunch.ShellScriptBuilder sb = ContainerLaunch.ShellScriptBuilder.create()
>> return Shell.WINDOWS?new WindowsShellScriptBuilder() : new UnixShellScriptBuilder()
    //在我们这个情景中是 UnixShellScriptBuilder
>>> UnixShellScriptBuilder.UnixShellScriptBuilder()
>>>> line("#!/bin/bash")
> if (environment != null) {
```

```

>+ for (Map.Entry<String,String> env : environment.entrySet()) {
>++ sb.env(env.getKey().toString(), env.getValue().toString()) == UnixShellScriptBuilder.env()
>+++> line("export ", key, " = \"", value, "\"") //为环境变量加 export 命令
>+ }
> }
> if (resources != null) {
>+ for (Map.Entry<Path,List<String>> entry : resources.entrySet()) {
>++ for (String linkName : entry.getValue()) {
>+++ sb.symlink(entry.getKey(), new Path(linkName)) == UnixShellScriptBuilder.symlink()
>++++> link(src, dst) == UnixShellScriptBuilder.link()
>++++>> line("ln -sf \"", src.toUri().getPath(), "\" \"", dst.toString(), "\"")
>+++ }
>+ }
> }
> sb.command(command) == UnixShellScriptBuilder.command()
>> line("exec /bin/bash -c \"", StringUtils.join(" ", command), "\"")
> PrintStream pout = new PrintStream(out) //在输出流 out 的基础上构建一个 PrintStream
> sb.write(pout) == UnixShellScriptBuilder.write()
//将 UnixShellScriptBuilder 中生成的各行语句写到脚本文件中
> out.close()

```

可见,这样写出去以后就是个 bash 脚本,其第一行为“#!/bin/bash”,下面是一些 export 命令,再下面也许有“ln -sf …”命令;但是脚本的核心无疑是“exec /bin/bash -c …”。这里的省略号代表来自 CLC 的命令行。

我们得记住,这个脚本是通过前面创建的输出流 containerScriptOutputStream 往外写的,这个输出流乃是建立在文件 nmPrivateContainerScriptPath 上,因而这就是所生成的脚本文件 launch_container。

这个脚本由 exec.launchContainer() 加以执行。在我们这个情景中,这里的 exec 是个 DefaultContainerExecutor 对象。调用这个函数时的语句如下:

```

exec.launchContainer(container, nmPrivateContainerScriptPath,
nmPrivateTokensPath, user, appIdStr, containerWorkDir, localDirs, logDirs)

```

对照 DefaultContainerExecutor.launchContainer() 的摘要,特别是其调用参数表,我们可以知道实参与形参的对应关系。显然,脚本文件 nmPrivateContainerScriptPath 就是这个函数的第二个形参,而且正好同名。这个函数的摘要经适当展开之后是这样:

```

[ContainerLaunch.call() > DefaultContainerExecutor.launchContainer()]

```

```

DefaultContainerExecutor .launchContainer(Container container,
Path nmPrivateContainerScriptPath, Path nmPrivateTokensPath,
String user, String appId, Path containerWorkDir,

```

```

        List<String> localDirs, List<String> logDirs)
> for (String sLocalDir : localDirs) {
>+ createDir(containerDir, dirPerm, true, user)
    //这些目录在 ContainerLaunch.call()中只是生成了路径而并未创建,现加以创建
> }
> createDir(tmpDir, dirPerm, false, user)
> Path launchDst = new Path(containerWorkDir, ContainerLaunch.CONTAINER_SCRIPT)
    //字符串 CONTAINER_SCRIPT 为"launch_container"
> copyFile(nmPrivateContainerScriptPath, launchDst, user)
    //复制脚本文件至工作目录下,文件名仍是"launch_container"
> LocalWrapperScriptBuilder sb =
    getLocalWrapperScriptBuilder(containerIdStr, containerWorkDir)
>> return new UnixLocalWrapperScriptBuilder(containerWorkDir)
>> super(containerWorkDir) == LocalWrapperScriptBuilder.LocalWrapperScriptBuilder()
>>> this.wrapperScriptPath = new Path(containerWorkDir,
    Shell.appendScriptExtension("default_container_executor"))
    //加上扩充名,就成为 default container executor.sh,余类推
>> this.sessionScriptPath = new Path(containerWorkDir,
    Shell.appendScriptExtension("default_container_executor_session"))
> Path pidFile = getPidFilePath(containerId)
> if (pidFile != null) {
>+ sb.writeLocalWrapperScript(launchDst, pidFile)
    == UnixLocalWrapperScriptBuilder.writeLocalWrapperScript(launchDst, pidFile)
>+> writeSessionScript(launchDst, pidFile) //写交互脚本
>+>> out = lfs.create(sessionScriptPath, EnumSet.of(CREATE, OVERWRITE)) //创建文件
    // sessionScriptPath 为"default_container_executor_session.sh"
>+>> pout = new PrintStream(out) //为这个文件建立输出流
>+>> pout.println("#!/bin/bash") //脚本的第一行为"#!/bin/bash"
>+>> pout.println()
>+>> pout.println("echo $$ >" + pidFile.toString() + ".tmp")
>+>> pout.println("/bin/mv -f" + pidFile.toString() + ".tmp" + pidFile)
>+>> String exec = Shell.isSetsidAvailable?"exec setsid" : "exec"
>+>> pout.println(exec + "/bin/bash \"\" + launchDst.toUri().getPath().toString() + "\"")
>+> super.writeLocalWrapperScript(launchDst, pidFile) //写外层脚本:
    == LocalWrapperScriptBuilder.writeLocalWrapperScript(launchDst, pidFile)
>+>> out = lfs.create(wrapperScriptPath, EnumSet.of(CREATE, OVERWRITE)) //创建文件
    // wrapperScriptPath 为"default_container_executor.sh"
>+>> pout = new PrintStream(out) //为这个文件建立输出流
>+>> writeLocalWrapperScript(launchDst, pidFile, pout)
    == UnixLocalWrapperScriptBuilder.writeLocalWrapperScript(launchDst, pidFile, pout)

```



```

>+>>> String exitCodeFile = ContainerLaunch.getExitCodeFile(pidFile.toString())
>+>>> String tmpFile = exitCodeFile + ".tmp"
>+>>> pout.println("#!/bin/bash") //脚本的第一行也是“#!/bin/bash”
>+>>> pout.println("/bin/bash \"\" + sessionScriptPath.toString() + "\"")
>+>>> pout.println("rc = $?")
>+>>> pout.println("echo $rc > \"\" + tmpFile + "\"")
>+>>> pout.println("/bin/mv -f \"\" + tmpFile + "\" \"\" + exitCodeFile + "\"")
>+>>> pout.println("exit $rc")
> } //end if(pidFile != null)
> setScriptExecutable(launchDst, user)
> setScriptExecutable(sb.getWrapperScriptPath(), user)
> Shell.CommandExecutor shExec = buildCommandExecutor(sb.getWrapperScriptPath().toString(),
    containerIdStr, user, pidFile, new File(containerWorkDir.toUri().getPath()),
    container.getLaunchContext().getEnvironment())
    //构建一个命令执行器
>> String[] command = getRunCommand(wrapperScriptPath, containerIdStr, user,
    pidFile, this.getConf()) //构建启动脚本的命令行
    == ContainerExecutor.getRunCommand()
>>> containerSchedPriorityAdjustment =
    YarnConfiguration.DEFAULT_NM_CONTAINER_EXECUTOR_SCHED_PRIORITY
    //优先级调整值为 0
>>> if (Shell.WINDOWS) { //宿主系统为 Windows
>>>+ return new String[] { Shell.WINUTILS, "task", "create", groupId, "cmd /c " + command }
>>> } else { //宿主系统为 Linux:
>>>+ retCommand = new ArrayList<String>()
>>>+ if (containerSchedPriorityIsSet) { //如果给定了优先级,就加上“nice - n...”
>>>++ retCommand.addAll(Arrays.asList("nice", "-n",
    Integer.toString(containerSchedPriorityAdjustment)))
>>>+ }
>>>+ retCommand.addAll(Arrays.asList("bash", command)) //在命令行前面加上“bash”
>>>+ return retCommand.toArray(new String[retCommand.size()]) //转换成 String[] 格式
>>> } // getRunCommand() 结束,命令行已转换成 String[] 格式
>> LOG.info("launchContainer: " + Arrays.toString(command))
>> return new ShellCommandExecutor(command, wordDir, environment)
    //返回后成为 shExec
> if (isContainerActive(containerId)) { //此前已经允许发起 Container 运行
>+ shExec.execute() == ShellCommandExecutor.execute() //让操作系统执行 shell 命令行
> }

```

这里简述一下这个函数的大致流程。

首先是把路径为 nmPrivateContainerScriptPath, 即名为 launch_container 的脚本文件复

制到工作目录下,文件名仍为 launch_container。

然后通过 getLocalWrapperScriptBuilder() 创建一个外层脚本构造器,其类型为 LocalWrapperScriptBuilder。但这只是个抽象类,所以在我们这个情景中实际创建的是 UnixLocalWrapperScriptBuilder,这是对抽象类 LocalWrapperScriptBuilder 的落实和扩充。要创建 UnixLocalWrapperScriptBuilder,当然要执行它的构造函数,而这个构造函数则创建了两个脚本文件的路径。其一是 wrapperScriptPath,即外层(包装)脚本的路径,这个脚本的文件名为 default_container_executor.sh。其二是 sessionScriptPath,即对话脚本的路径,其文件名为 default_container_executor_session.sh。注意这只是两个路径名,此时尚未创建实际的脚本文件。

接着,如本容器的 pid 文件路径已存在,就调用该构造器即 UnixLocalWrapperScriptBuilder 的 writeLocalWrapperScript(),这个函数创建了上述 SessionScript 和 LocalWrapperScript 两个脚本文件,并生成两个脚本的内容。

这两个脚本,连同本已存在的脚本 launch_container,三个脚本之间的关系是这样的:首先启动的是外层脚本 default_container_executor.sh,这个外层脚本会启动执行“会议脚本” default_container_executor_session.sh,而会议脚本会启动执行脚本 launch_container,脚本 launch_container 中则含有来自 CLC 的命令行。

做好了这些准备工作之后,就通过 buildCommandExecutor() 构建一个实现了 Shell.CommandExecutor 界面的命令执行器 ShellCommandExecutor 对象。注意,这里的 Shell 是 Hadoop 代码中定义的一个抽象类,并不直接就是操作系统的 shell。而 ShellCommandExecutor 则是其内部定义的一个类,是对 Shell 的扩充。调用这个函数时的第一个参数就是 getWrapperScriptPath(),那就是外层脚本文件的路径。创建了命令执行器之后,就调用其 execute() 函数执行预定的脚本,那就是 default_container_executor。而 default_container_executor 脚本中则有类似于这样的命令行(这里只写了文件名而不是全路径名):

```
/bin/bash default_container_executor_session.sh
```

这会启动脚本 default_container_executor_session 的执行,那个脚本中则有类似于这样的命令行:

```
/bin/bash launch_container
```

在脚本 launch_container 中则有来自 CLC 中的命令行,那才是我们真正的目的所在。

我们不妨看一下实际运行时所生成的日志。在我的机器上,运行以后日志文件中有类似于这样的记录:

```
...nodemanager.DefaultContainerExecutor: launchContainer: [bash /tmp/hadoop-root  
/nm-local-dir/usercache/root/appcache/application_1426734909742_0001  
/container_1426734909742_0001_01_000001/default_container_executor.sh]
```

因为我没有为命令行设置优先级,所以这里没有“nice -n”这样的选项。显然,这只是用 bash 运行外层脚本文件 default_container_executor.sh。这个外层脚本会启动执行会议脚本 default_container_executor_session.sh,再下一层才是容器脚本 launch_container。

真正要启动执行的命令行则在容器脚本文件 `launch_container` 中,那个脚本的核心就是由 `ContainerLaunchContext` 带来的命令行“`~/bin/java ... MRAppMaster ...`”,就是要在宿主操作系统上运行 `bin/java`,即 Java 虚拟机,让其执行 `MRAppMaster.class`。

运行 `MRAppMaster.class` 的 JVM 一经启动,当地就有了一个独立的 JVM 进程,这个 JVM 从 `MRAppMaster.main()` 开始执行,这就是为所提交 App 在 Hadoop 集群上成立的“项目组”的组长。再往后就是 `MRAppMaster` 的事了。

后面我们还将看到,这样的“项目组长”其实并不一定是 `MRAppMaster`,而也可以是 `ApplicationMaster`。`MRAppMaster` 只是 MapReduce 计算的项目领导者,`ApplicationMaster` 则是类似于 Unix/Linux 风格的流水线式计算的项目领导者。当然,目前在 Hadoop 集群上最常见的计算还是 MapReduce。

另外,这里我们要在目标 NM 节点上投运的容器只是用于为 App 创建运行着 `MRAppMaster` 或其他 AppMaster 的 JVM 进程。但是,同样的过程也可用于创建其他进程,由 `ContainerLaunchContext` 带来 NM 节点上的最核心的信息就是个 shell 命令行。

第 8 章

MRAppMaster 与作业投运

8.1 MRAppMaster

在上一章中我们看到,用户向 ResourceManager 提交作业(或称 App)以后,RM 会选择一个 NodeManager 节点,让其提供一定的资源,担任类似于项目组长的角色,然后将与作业有关的资源信息打包在一个容器里,再和作业的其他有关信息一起封装在一个 ContainerLaunchContext 对象中发送给这个节点,经过资源本地化之后就在该节点的宿主操作系统上执行一个命令行,发起一个 JVM 进程,让这个 Java 虚拟机执行 MRAppMaster.class。MRAppMaster,意为“执行 MapReduce 计算的 AppMaster”,是专门面向 MapReduce 计算的。可想而知,这是个带有 main()函数、可以作为独立 JVM 进程运行的类。

MRAppMaster 类对象起着类似于项目组长的作用,其职责是:根据 ContainerLaunchContext 中的信息将作业分解为若干任务,例如 16 个 Mapper、1 个 Reducer,不用 Combiner,那就是 17 个任务,然后与 RM 节点协商,为这些任务分配资源。为任务分配资源的过程也就是把任务指派到某些节点上的过程。一方面,显然只有 CPU 虚核和内存有足够空闲的节点才能接受任务指派;另一方面,数据也是重要资源,计算应该尽量靠近数据,还得考虑因此而引起的网络流量和延迟。另外,把许多任务指派到一群节点上,实际上也是搭建数据流(或工作流)的过程。进一步,把具体的任务指派到目标节点上后,那里也会有个资源本地化的过程,会有容器的投运。

不过,更准确地说,资源并不是分配给具体的任务,而是分配给为执行具体任务所进行的尝试。比方说,一个任务的第一次尝试失败了,但是再进行一次尝试就成功了,这先后两次的尝试都需要为其分配资源。所以,资源分配的受主是执行任务的尝试,而不是任务本身。执行某个具体任务的一次尝试,就称为对此任务的一次“任务尝试”,即 TaskAttempt。同理,把任务(Task)指派到节点上,更准确地说只是把 TaskAttempt 指派到节点上。

在 YARN 子系统中,NodeManager 节点的计算能力不是用于如 MRAppMaster 那样对 App 的任务管理,就是用于执行构成 App 的具体任务,或者二者兼有,要不然就是空转了。所以,MRAppMaster 也是 NodeManager 节点上的重要活动之一。

我们先看一下 MRAppMaster 类的摘要:

```
class MRAppMaster extends CompositeService {  
    > String appName  
    > ApplicationAttemptId appAttemptID//MRAppMaster 对象所代表的 ApplicationAttempt
```

```

> ContainerId containerID    //所投运的容器
> String nmHost              //所在节点的主机名
> int nmPort
> int nmHttpPort
> AppContext context
> Dispatcher dispatcher
> ClientService clientService
> ContainerAllocator containerAllocator
> ContainerLauncher containerLauncher
> EventHandler<CommitterEvent> committerEventHandler
> Speculator speculator
> TaskAttemptListener taskAttemptListener
> ClassLoader jobClassLoader
> OutputCommitter committer
> JobEventDispatcher jobEventDispatcher
> ... //别的 Dispatcher
> Job job
] main()
] serviceInit(final Configuration conf)
] serviceStart() //执行完 serviceInit(),就轮到 serviceStart()了
] class ContainerAllocatorRouter{}
] class ContainerLauncherRouter{}
] class RunningAppContext implements AppContext {}
] class JobEventDispatcher{}
] class TaskEventDispatcher{}
] class TaskAttemptEventDispatcher{}
] class SpeculatorEventDispatcher{}

```

定义于 MRAppMaster 内部的那些类,我们随着情景的发展会看到它们的作用。现在我们跟随着 Java 虚拟机的启动,从 MRAppMaster 的 main()函数开始看它的流程:

```

MRAppMaster.main()
> containerIdStr = System.getenv(Environment.CONTAINER_ID.name())
                                //通过环境变量 CONTAINER_ID 传递 containerId 字符串
> containerId = ConverterUtils.toContainerId(containerIdStr)
> applicationAttemptId = containerId.getApplicationAttemptId()
> appMaster = new MRAppMaster(applicationAttemptId, containerId, nodeHostString, ...)
>> MRAppMaster(ApplicationAttemptId applicationAttemptId,
    ContainerId containerId, String nmHost, int nmPort, int nmHttpPort,
    Clock clock, long appSubmitTime) // MRAppMaster 类的构造函数
> ShutdownHookManager.get().addShutdownHook(

```

```

        new MRAppMasterShutdownHook(appMaster), ...)
        //加上关机挂钩 MRAppMasterShutdownHook,使其在关机时得到调用
> JobConf conf = new JobConf(new YarnConfiguration()) //重新创建 JobConf
    //配置信息主要来自“yarn-default.xml”、“yarn-site.xml”、“core-site.xml”等文件
> conf.addResource(new Path(MRJobConfig.JOB_CONF_FILE)) //“job.xml”
> initAndStartAppMaster(appMaster, conf, jobUserName)
>> UserGroupInformation appMasterUgi =
    UserGroupInformation.createRemoteUser(jobUserName)
>> appMasterUgi.doAs(new PrivilegedExceptionAction<Object>()) {} //用户权限控制
    ] run()
    > appMaster.init(conf)
    >> serviceInit(final Configuration conf) //先执行 serviceInit()
    > appMaster.start()
    >> serviceStart() //再执行 serviceStart()

```

MRAppMaster 是奉 RM 之命由 NodeManager 在其所在 NM 节点上创建的,但是真正的委托人却是作业的提交者,MRAppMaster 的权限不应高于作业的委托人,所以这里先通过 `createRemoteUser()` 获取相关信息并创建代表着作业提交者身份的 UGI 即 `appMasterUgi`,并通过 `doAs()` 以委托人的身份执行 MRAppMaster 的初始化并开始运行。

```
[initAndStartAppMaster() > run() > MRAppMaster.serviceInit()]
```

```

MRAppMaster.serviceInit(Configuration conf)
> createJobClassLoader(conf)
> context = new RunningAppContext(conf)
> conf.setInt(MRJobConfig.APPLICATION_ATTEMPT_ID, appAttemptID.getAttemptId())
> numReduceTasks = conf.getInt(MRJobConfig.NUM_REDUCES, 0) //获取 Reducer 的数量
> committer = createOutputCommitter(conf)
>> action = new Action<OutputCommitter>() {}
    //创建一个实现了 Action 界面的对象,动态定义其 call() 函数
    ] call(Configuration conf)
        > taskID = MRBuilderUtils.newTaskId(jobId, 0, TaskType.MAP)
        > attemptID = MRBuilderUtils.newTaskAttemptId(taskID, 0)
        > taskContext = new TaskAttemptContextImpl(conf,
            TypeConverter.fromYarn(attemptID))
        > OutputFormat outputFormat =
            ReflectionUtils.newInstance(taskContext.getOutputFormatClass(), conf)
        > OutputCommitter committer = outputFormat.getOutputCommitter(taskContext)
        > return committer
>> return callWithJobClassLoader(conf, action)
>>> ClassLoader currentClassLoader = conf.getClassLoader()

```



```

>>> action.call(conf) //调用上面那个 call 函数
>>> MRApps.setClassLoader(currentClassLoader, conf)
> dispatcher = createDispatcher() //事件分发器,类似于事件信息的路由器
>> return new AsyncDispatcher() //MRAppMaster 的 Dispatcher 是 AsyncDispatcher
> clientService = createClientService(context) //service to handle requests from JobClient
>> return new MRClientService(context) //创建 MRClientService
> clientService.init(conf) //并调用其 init()
> containerAllocator = createContainerAllocator(clientService, context)
>> new ContainerAllocatorRouter(clientService, context) //创建 ContainerAllocatorRouter
> committerEventHandler = createCommitterEventHandler(context, committer)
>> new CommitterEventHandler(context, committer,
                                getRMHeartbeatHandler(), jobClassLoader)
> taskAttemptListener = createTaskAttemptListener(context)
>> new TaskAttemptListenerImpl(context, jobTokenSecretManager, getRMHeartbeatHandler())
> this.jobEventDispatcher = new JobEventDispatcher()
> dispatcher.register(JobEventType.class, jobEventDispatcher)
    //向 dispatcher 登记由 jobEventDispatcher 接收,类型为 JobEventType 的事件,下同
> dispatcher.register(TaskEventType.class, new TaskEventDispatcher())
> dispatcher.register(TaskAttemptEventType.class, new TaskAttemptEventDispatcher())
> dispatcher.register(CommitterEventType.class, committerEventHandler)
> if (conf.getBoolean(MRJobConfig.MAP_SPECULATIVE, false)
>+ speculator = createSpeculator(conf, context)
> speculatorEventDispatcher = new SpeculatorEventDispatcher(conf)
> dispatcher.register(Speculator.EventType.class, speculatorEventDispatcher)
> // service to allocate containers from RM (if non-uber) or to fake it (uber)
> dispatcher.register(ContainerAllocator.EventType.class, containerAllocator)
> containerLauncher = createContainerLauncher(context)
>> new ContainerLauncherRouter(context)
> dispatcher.register(ContainerLauncher.EventType.class, containerLauncher)
> super.serviceInit(conf)

```

这里创建了一个事件分发器 dispatcher,这就好比一台路由器,下面向 dispatcher 登记了多种事件的接受者,就好比配置了许多路由。这样就相当于构成了 MRAppMaster 内部的一个事件信息网,为 MRAppMaster 内部各个部件的状态机运行做好了准备。

执行完 MRAppMaster.serviceInit(),下面就是 MRAppMaster.serviceStart()。

```
[initAndStartAppMaster() > run() > MRAppMaster.serviceStart()]
```

```

MRAppMaster.serviceStart()
> processRecovery()
> Job job = createJob(getConfig(), forcedState, shutDownMessage)

```

```

>>> Job newJob = new JobImpl(jobId, appAttemptID, conf, dispatcher.getEventHandler(), ...)
>>> ((RunningAppContext) context).jobs.put(newJob.getID(), newJob)
>>> dispatcher.register(JobFinishEvent.Type.class, createJobFinishEventHandler())
> DefaultMetricsSystem.initialize("MRAppMaster") //用于统计
> JobEvent initJobEvent = new JobEvent(job.getID(), JobEventType.JOB_INIT)
> jobEventDispatcher.handle(initJobEvent)
    //向 JobImpl 发送 JOB_INIT 事件,使其状态机进入 INITED 状态
> clientService.start() == MRClientService.start()
>>> MRClientService.serviceStart()
>>>> address = new InetSocketAddress(0)
>>>> server = rpc.getServer(MRClientProtocol.class, protocolHandler, address, ...)
>>>> server.start()
>>>> this.bindAddress = NetUtils.createSocketAddrForHost(appContext.getNMHostname(),
    server.getListenerAddress().getPort())
>>>> webApp = WebApps.$for("mapreduce", ...) //创建 Web 服务,见本书第 18 章
>>>> super.serviceStart()
> super.serviceStart()
> MRApps.setClassLoader(jobClassLoader, getConfig())
> startJobs() // * * create a job - start event to get this ball rolling * /
>>> startJobEvent = new JobStartEvent(job.getID(), recoveredJobStartTime)
>>>> super(jobID, JobEventType.JOB_START)
>>> // * * send the job - start event.this triggers the job execution. * /
>>> dispatcher.getEventHandler().handle(startJobEvent) //向 JobImpl 发送 JOB_START 事件

```

MRAppMaster 本来就是为具体的作业而建的,所以这里要创建一个 JobImpl 对象,代表着这个 MRAppMaster 要为之奋斗的作业。作业的部署和投运是个过程,所以 JobImpl 对象内部有个状态机。这里接连向其发送了两个 JobEventType 事件,第一个是 JOB_INIT,然后是 JOB_START。我们刚才看到,JobEventType 事件的接受者是 jobEventDispatcher,所以 dispatcher 会把这个事件转给 jobEventDispatcher。不过,对此过程我们现在已经不关心了,我们关心的是 JobImpl 状态机对此的反应:

```

addTransition (JobStateInternal.NEW,
    EnumSet.of(JobStateInternal.INITED, JobStateInternal.NEW),
    JobEventType.JOB_INIT, new InitTransition())

```

这就是,由于状态机的当前状态是 NEW,所以在受 JOB_INIT 事件触发时会执行 InitTransition.transition(),并视返回结果进入 INITED 或 NEW 状态。

```

[initAndStartAppMaster() > run() > MRAppMaster.serviceStart() > JobEventType.JOB_INIT
=> InitTransition.transition()]

```

```

JobImpl.InitTransition.transition(JobImpl job, JobEvent event)
> if (job.newApiCommitter) //如果采用新 API:

```

```

>+ job.jobContext = new JobContextImpl(job.conf, job.oldJobId)
> else //如果采用老 API
>+ job.jobContext = new org.apache.hadoop.mapred.JobContextImpl(job.conf, job.oldJobId)
> setup(job)
> job.fs = job.getFileSystem(job.conf)
> JobSubmittedEvent jse = new JobSubmittedEvent(job.oldJobId, ...) //log to job history
> job.eventHandler.handle(new JobHistoryEvent(job.jobId, jse)) //我们不关心 HistoryServer
> TaskSplitMetaInfo[] taskSplitMetaInfo = createSplits(job, job.jobId) //获取分片信息
>> TaskSplitMetaInfo[] allTaskSplitMetaInfo =
        SplitMetaInfoReader.readSplitMetaInfo(job.oldJobId, job.fs,
        job.conf, job.remoteJobSubmitDir)
>> return allTaskSplitMetaInfo //输入文件分片信息来自 JobImpl 的数据结构部分
> job.numMapTasks = taskSplitMetaInfo.length //MapTask 的数量取决于输入文件分片
> job.numReduceTasks = job.conf.getInt(MRJobConfig.NUM_REDUCE, 0)
        //ReduceTask 的数量根据设定,默认为 0
> if (job.numMapTasks == 0) job.reduceWeight = 0.9f
> else if (job.numReduceTasks == 0) job.mapWeight = 0.9f
> job.mapWeight = job.reduceWeight = 0.45f
> for (int i = 0; i < job.numMapTasks; ++i) {
>+ inputLength += taskSplitMetaInfo[i].getInputDataLength() //统计输入数据总长度
> }
> job.makeUberDecision(inputLength) //根据输入数据总长度决定是否采用 Uber 模式
> job.taskAttemptCompletionEvents = new ArrayList<TaskAttemptCompletionEvent>(
        job.numMapTasks + job.numReduceTasks + 10)
> job.mapAttemptCompletionEvents =
        new ArrayList<TaskCompletionEvent>(job.numMapTasks + 10)
> job.taskCompletionIdxToMapCompletionIdx =
        new ArrayList<Integer>(job.numMapTasks + job.numReduceTasks + 10)
> job.allowedMapFailuresPercent =
        job.conf.getInt(MRJobConfig.MAP_FAILURES_MAX_PERCENT, 0)
> job.allowedReduceFailuresPercent =
        job.conf.getInt(MRJobConfig.REDUCE_FAILURES_MAXPERCENT, 0)
> createMapTasks(job, inputLength, taskSplitMetaInfo)
        //create the Tasks but don't start them yet
>> for (int i = 0; i < job.numMapTasks; ++i) {
>>+ TaskImpl task = new MapTaskImpl(job.jobId, i, job.eventHandler, ..., job.conf, splits[i], ...)
>>+ job.addTask(task)
>> }
> createReduceTasks(job)
>> for (int i = 0; i < job.numReduceTasks; i++) {

```

```
>>>+ TaskImpl task = new ReduceTaskImpl(job.jobId, i, ...,
                                   job.conf, job.numMapTasks, job.taskAttemptListener, ...)
>>>+ job.addTask(task)
>>> }
> return JobStateInternal.INITED
```

参数 job 是个 JobImpl 类对象,这个对象中有关于具体作业的种种信息,特别是里面有个成分就是 JobContext。当然,这些信息都来自作业提交。

这里首先要做的就是作业的分解,根据作业的要求创建出分别代表着 MapTask 和 ReduceTask 的众多 MapTaskImpl 对象和 ReduceTaskImpl 对象。这些对象将被分发到具体执行计算的 NM 节点上,但是现在还早,还有较长的一段路程要走。

特别要注意这里通过 createSplits() 创建的 TaskSplitMetaInfo 数组 taskSplitMetaInfo。这是本作业数据输入分片的信息,是由 readSplitMetaInfo() 从用户所提交的 Split 元数据文件里读入的。前面讲过,当 AM 向 RM 申请分配容器和资源时, RM 节点上的资源调度器怎么知道应该把一个作业的各个任务放在哪些节点上运行呢? 原则是“数据在哪里,计算就在哪里进行”。可是,怎么知道数据在哪里呢? 这个问题的谜底就在这里。这个 JobImpl 对象就在 AM 中。有了 taskSplitMetaInfo 这个数组,就知道了输入文件一共分成几个 Split, 这些 Split 各自的数据(输入文件中的数据块复份)又在哪些节点上。有了这些信息, AM 向 RM 节点申请分配容器和资源的时候就可以指定节点和机架了。另外,这个数组也直接决定了 MapTask 的创建,输入数据分几个片, createMapTasks() 就创建几个 MapTask。ReduceTask 的数量则是由用户在程序中设定的。

要是顺利完成这些操作而并未发生异常,就返回 JobStateInternal.INITED, 因而这个 JobImpl 状态机就进入 INITED 状态。

然后,当 MRAppMaster.serviceStart() 执行到末尾的时候,又向 JobImpl 发来了 JobEventType.JOB_START 事件,而 JobImpl 状态机对此的反应为:

```
addTransition(JobStateInternal.INITED, JobStateInternal.SETUP,
              JobEventType.JOB_START, new StartTransition())
```

即先执行 StartTransition.transition(), 然后转入 SETUP 状态。

```
[initAndStartAppMaster() > run() > MRAppMaster.serviceStart() > JobEventType.JOB_START
=> StartTransition.transition()]
```

```
JobImpl.StartTransition.transition(JobImpl job, JobEvent event)
> jse = (JobStartEvent) event
> JobInitedEvent jie = new JobInitedEvent(job.oldJobId, job.startTime,
                                   job.numMapTasks, job.numReduceTasks, ...)
> job.eventHandler.handle(new JobHistoryEvent(job.jobId, jie))
> JobInfoChangeEvent jice = new JobInfoChangeEvent(job.oldJobId,
                                   job.appSubmitTime, job.startTime)
> job.eventHandler.handle(new JobHistoryEvent(job.jobId, jice))
```

//上面几个事件都是为运行历史记录而发的

```
> e = new CommitterJobSetupEvent(job.jobId, job.jobContext)
>> super(CommitterEventType.JOB_SETUP)
> job.eventHandler.handle(e) //由 CommitterEventHandler.handle()加以处理
    == CommitterEventHandler.handle(CommitterEvent event)
>> eventQueue.put(event)
```

这里前面几个事件都是为运行日志和历史记录而发的,不在我们关心的“主旋律”中,但是最后的 CommitterEventType.JOB_SETUP 是发送给一个 CommitterEventHandler 对象的,这就重要了。这个 CommitterEventHandler 对象是在前面 MRAppMaster.serviceInit()中创建的。这个类的摘要如下:

```
class CommitterEventHandler extends AbstractService
    implements EventHandler<CommitterEvent> {}

] serviceInit(Configuration conf)
] serviceStart()

> tfBuilder = new ThreadFactoryBuilder()
    .setNameFormat("CommitterEvent Processor # %d")

> if (jobClassLoader != null) {
>+ backingTf = new ThreadFactory()
    ] newThread(Runnable r)
        > thread = new Thread(r)
        > thread.setContextClassLoader(jobClassLoader)
        > return thread
>+ tfBuilder.setThreadFactory(backingTf)
> }

> ThreadFactory tf = tfBuilder.build() //创建 ThreadFactory
> launcherPool = new ThreadPoolExecutor(5, 5, 1, TimeUnit.HOURS,
    new LinkedBlockingQueue<Runnable>(), tf)
    //创建线程池

> eventHandlingThread = new Thread(new Runnable()) //创建线程
    ] run()
        > while (!stopped.get() && !Thread.currentThread().isInterrupted()) {
        >+ event = eventQueue.take() //从队列中摘下一个事件
        >+ launcherPool.execute(new EventProcessor(event))
            //拿到一个事件就另起一个 CommitterEventHandler.EventProcessor 线程
        > }

> eventHandlingThread.setName("CommitterEvent Handler")
> eventHandlingThread.start() //开始执行线程 eventHandlingThread 的 run()函数
> super.serviceStart()
```

CommitterEventHandler.handle()把前面 StartTransition.transition()发出的JOB_SETUP事

件挂入 eventQueue 队列,而 CommitterEventHandler.eventHandlingThread 则从该队列中摘下这个事件,另起一个 CommitterEventHandler.EventProcessor 线程来处理这个事件。

```
[StartTransition.transition() > CommitterEventType.JOB_SETUP
=> CommitterEventHandler.EventProcessor.run()]
```

```
CommitterEventHandler.EventProcessor.run()
> switch (event.getType()) {
> case JOB_SETUP:    //现在要处理的是 JOB_SETUP 事件
>+ handleJobSetup((CommitterJobSetupEvent) event)
>+> committer.setupJob(event.getJobContext()) == OutputCommitter.setupJob()
>+>> FileOutputCommitter.setupJob() //设置好本作业所有输出文件所在的目录
>+> e = new JobSetupCompletedEvent(event.getJobID())
>+>> super(jobID, JobEventType.JOB_SETUP_COMPLETED)
>+> context.getEventHandler().handle(e) //向 JobImpl 发送 JOB_SETUP_COMPLETED 事件
> case JOB_COMMIT:
>+ handleJobCommit((CommitterJobCommitEvent) event)
> case JOB_ABORT:
>+ handleJobAbort((CommitterJobAbortEvent) event)
> case TASK_ABORT:
>+ handleTaskAbort((CommitterTaskAbortEvent) event)
> }
```

程序上这儿来绕了一下,为的只是设置本作业输出文件的存放目录,这个输出目录就是用户在提交作业前在程序中通过 FileOutputFormat.setOutputPath(job, outDir) 设置的目录。读者可以回到例如 examples/QuasiMonteCarlo.java 中看一下。设置好以后,就向 JobImpl 发送一个 JOB_SETUP_COMPLETED 事件,“球”又回到了 JobImpl 那儿:

```
addTransition(JobStateInternal.SETUP, JobStateInternal.RUNNING,
               JobEventType.JOB_SETUP_COMPLETED, new SetupCompletedTransition())
```

JobImpl 状态机的反应是先执行 SetupCompletedTransition.transition(), 然后转入 RUNNING 状态。

```
JobImpl.SetupCompletedTransition.transition(JobImpl job, JobEvent event)
> job.scheduleTasks(job.mapTasks, job.numReduceTasks == 0)
> job.scheduleTasks(job.reduceTasks, true)
>> for (TaskId taskID : taskIDs) {
>>+ TaskInfo taskInfo = completedTasksFromPreviousRun.remove(taskID)
>>+ if (taskInfo != null) { //原先就有,是要恢复运行的老任务
>>++ e = new TaskRecoverEvent(taskID, taskInfo, committer, recoverTaskOutput)
>>++> super(taskID, TaskEventType.T_RECOVER)
>>++ eventHandler.handle(e)
```



```

>>>+ } else { //原先没有,这是要调度运行的新任务
>>>++ e = new TaskEvent(taskID, TaskEventType.T_SCHEDULE)
>>>++ eventHandler.handle(e) //向具体的 TaskImpl 发送 T_SCHEDULE 事件
>>>+ }
>>> }

```

先对本作业的所有 Map 任务调用 JobImpl.scheduleTasks(),再对本作业的所有 Reduce 任务调用 JobImpl.scheduleTasks()。一般这些任务都是新建的,并无上一次运行(PreviousRun)和 TaskRecover 可言,但是有时候也可能会有那样的情况。这两种情况下向目标任务即 TaskImpl 发出的事件不一样;但总是需要向这 TaskImpl 发送一个事件,这一点是一样的。

一个作业通常会有很多任务,这里是对所有这些任务的循环。再往下一层,就是对其中各个具体任务即 TaskImpl 的考察和处理了。

对于新创建的 TaskImpl,向其发送的是 T_SCHEDULE 事件。不过 TaskImpl 是抽象类,具体落地的是 MapTaskImpl 或 ReduceTaskImpl。所以,具体的 TaskImpl 对象是前面通过 createMapTasks() 和 createReduceTasks() 创建的 MapTaskImpl 或 ReduceTaskImpl,二者都是对 TaskImpl 的扩充,有着同样的状态机。它们对 T_SCHEDULE 事件的反应是:

```

addTransition(TaskStateInternal.NEW, TaskStateInternal.SCHEDULED,
              TaskEventType.T_SCHEDULE, new InitialScheduleTransition())

```

先执行 InitialScheduleTransition.transition(),然后转入 SCHEDULED 状态。

```

[SetupCompletedTransition.transition() > TaskEventType.T_SCHEDULE
=> InitialScheduleTransition.transition()]

```

```

TaskImpl.InitialScheduleTransition.transition(TaskImpl task, TaskEvent event)
> task.addAndScheduleAttempt(Avataar.VIRGIN) //增加一个初次尝试,并加以调度
//VIRGIN 表示原初,SPECULATIVE 表示替补
>>> TaskAttempt attempt = addAttempt(avataar)
>>>> TaskAttemptImpl attempt = createAttempt() //创建具体的 TaskAttemptImpl
>>>> attempt.setAvataar(avataar)
>>>> LOG.debug("Created attempt " + attempt.getID())
>>>> switch (attempts.size()) {
>>>> case 0:
>>>>+ attempts = Collections.singletonMap(attempt.getID(), (TaskAttempt) attempt)
//本任务尚未创建 attempts,这个 attempt 是第一个,先为其创建一个临时 MAP
>>>> case 1:
>>>>+ Map<TaskAttemptId, TaskAttempt> newAttempts =
new LinkedHashMap<TaskAttemptId, TaskAttempt>(maxAttempts)
//刚创建的这个 attempt 是第二个,创建一个正式的 Map 来取代临时 Map

```

```

>>>+ newAttempts.putAll(attempts)    //从原先的临时 Map 转移过来
>>>+ attempts = newAttempts          //取代
>>>+ attempts.put(attempt.getID(), attempt) //将刚创建的 attempt 也加进去
>>> default;
>>>+ attempts.put(attempt.getID(), attempt) //将 attempt 放入正式的 TaskImpl.attempts
>>> }
>>> ++ nextAttemptNumber
>>> return attempt
>> if (failedAttempts.size() > 0) { //曾有失败的 attempt,需要调度重试。
>>+ e = new TaskAttemptEvent(attempt.getID(), TaskAttemptEventType.TA_RESCHEDULE)
>>+ eventHandler.handle(e)
>> } else { //无失败记录,直接调度
>>+ e = new TaskAttemptEvent(attempt.getID(), TaskAttemptEventType.TA_SCHEDULE)
>>+ eventHandler.handle(e) //向 TaskAttempt 发送 TA_SCHEDULE 事件
>> }
> task.scheduledTime = task.clock.getTime()
> task.sendTaskStartedEvent() //为历史记录发送任务启动事件

```

这个 TaskImpl 状态机的这次跳变称为 InitialScheduleTransition,显然是“初始调度”的意思。但是,我们知道,需要被调度运行的不是这个任务本身,而是对于执行这个任务的一次尝试。所以这里要 addAndScheduleAttempt(),要添加一个 TaskAttempt,实际上是 TaskAttemptImpl,就是程序中的 attempt。创建时的参数 Avataar.VIRGIN 表示这是一次原初尝试,而不是替补尝试。

TaskImpl 维持一个关于 TaskAttemptImpl 的 Map,使得凭 attemptID 就可从中查到其 TaskAttemptImpl 对象。在大多数情况下,具体任务都是一次尝试就成功的,这样的 Map 当然比较简单。只有在屡败屡试的情况下这个 Map 中才会有多个 TaskAttemptImpl 存在。所以,这里对于 Map 的处理貌似复杂,其实只是对于运行效率的优化。

最后,根据此前是否仍有失败的尝试尚未解决,对具体的 TaskImpl 发送或为 TA_RESCHEDULE 或为 TA_SCHEDULE 的 TaskAttemptEvent 事件。当然,绝大多数情况下这里发出的都是 TA_SCHEDULE,因为失败的任务尝试毕竟是少数,所以我们在这里将只关心 TA_SCHEDULE。

8.2 App 资源与容器

为本任务创建了代表着一次原初尝试的 TaskAttemptImpl 对象之后,TaskImpl 即向其发出一个 TA_SCHEDULE 事件,而 TaskAttemptImpl 状态机对此的反应则是:

```

addTransition(TaskAttemptStateInternal.NEW, TaskAttemptStateInternal.UNASSIGNED,
              TaskAttemptEventType.TA_SCHEDULE, new RequestContainerTransition(false))

```

事件 TA_SCHEDULE 实际上是个调度请求,所以 TaskAttemptImpl 对此的反应是调用 RequestContainerTransition.transition(),要求为这次尝试分配容器,然后进入 UNASSIGNED 状

态,表示已经要求分配指派,但尚未指派。所谓容器,里面当然包含种种的资源,但是其中最关键的还是 VCore 和内存,这二者都在具体的节点上,不像别的资源那样可以搬迁,可以“本地化”。所以,分配到一个容器,实际上就是被指派到了某个节点上。这就好像从前的大学生毕业后一经“分配工作”便要前去报到一样。

首先要申请分配容器:

```
[SetupCompletedTransition.transition() > TaskEventType.T_SCHEDULE
=> InitialScheduleTransition.transition()>TaskAttemptEventType.TA_SCHEDULE
=> RequestContainerTransition.transition()]
```

```
TaskAttemptImpl.RequestContainerTransition.transition()
> // Tell any speculator that we're requesting a container
> e = new SpeculatorEvent(taskAttempt.getID().getTaskId(), +1)
>> super(Speculator.EventType.TASK_CONTAINER_NEED_UPDATE, timestamp)
> taskAttempt.eventHandler.handle(e)
           //由 SpeculatorEventDispatcher 交由 DefaultSpeculator 处理
> //request for container
> if (rescheduled) { //这是尝试失败后另行申请分配容器
>+ e = ContainerRequestEvent.createContainerRequestEventForFailedContainer(
           taskAttempt.attemptId, taskAttempt.resourceCapability)
>+ taskAttempt.eventHandler.handle(e)
> } else {           //这是常规任务尝试的容器申请
>+ e = new ContainerRequestEvent(taskAttempt.attemptId, taskAttempt.resourceCapability, ...)
>+> super(attemptID, ContainerAllocator.EventType.CONTAINER_REQ)
>+ taskAttempt.eventHandler.handle(e) //发送给 containerAllocator
> }
```

申请分配容器有两种情况:一种是任务尝试失败之后要求另行分配一个容器,换个地方再去试试;另一种是原初的任务尝试,要求分配一个容器、指定一个节点。但是,再次尝试之前还要向一个“替补者 Speculator”,通常是 DefaultSpeculator,发送一个事件,起着打个招呼的作用。这是 MRAppMaster 在其 serviceInit()阶段通过 createSpeculator()创建的。

当然,关键的操作还是发送 CONTAINER_REQ 事件。这个事件的发送目标值得一提。程序中的 taskAttempt.eventHandler并非这个事件的发送目标,它只是将事件交付给 Dispatcher,就像把信送到邮局。那么谁是这个事件的接收者和处理者呢?

MRAppMaster.serviceInit()中有这么几行代码:

```
// service to allocate containers from RM (if non-uber) or to fake it (uber)
containerAllocator = createContainerAllocator(null, context);
addIfService(containerAllocator);
dispatcher.register(ContainerAllocator.EventType.class, containerAllocator);
```

注释中所说的 service 就是指 containerAllocator。其作用是:从 RM 分配容器(如果不是

uber 模式),或者“伪造(fake)”出容器(如果是 uber 模式)。从代码中看,这里创建了一个 containerAllocator,并向 dispatcher 登记接收 ContainerAllocator.EventType 类的事件,所以这个 CONTAINER_REQ 事件的接收者应该就是 containerAllocator。

然而实际上 createContainerAllocator()所创建的是个 ContainerAllocatorRouter 对象,顾名思义这也只是个“路由器”,而并非真正的接受者和处理者。那么究竟谁是真正的接受者和处理者呢?我们看一下 ContainerAllocatorRouter 的代码。

```
] class ContainerAllocatorRouter{}
[] serviceStart()
    > if (job.isUber()) this.containerAllocator = new LocalContainerAllocator(...)
    > else this.containerAllocator = new RMContainerAllocator(...)
    > ((Service)this.containerAllocator).init(getConfig())
    > ((Service)this.containerAllocator).start()
```

就是说,在 MRAppMaster 创建 ContainerAllocator 的时候,实际创建的是哪一种取决于作业是否运行于 Uber 模式,而且一旦创建之后就固定下来了。所谓 Uber 模式,就是“优步”、“拼车”模式,让一个作业的所有任务都拼在同一节点上运行,MRAppMaster 在哪里,这些任务就在哪里运行。可想而知,这两种容器分配的复杂程度大不一样,本地的容器分配肯定比 RM 的容器分配简单。

顺便说一下,用来发起容器运行的 ContainerLauncher 也与此相似:创建 ContainerLauncher 对象时实际创建的是 ContainerLauncherRouter,后者会根据是否工作于 Uber 模式最终或创建 LocalContainerLauncher,或创建 ContainerLauncherImpl。

```
] class ContainerLauncherRouter{}
[] serviceStart()
    > if (job.isUber()) this.containerLauncher = new LocalContainerLauncher(...)
    > else this.containerLauncher = new ContainerLauncherImpl(context)
    > ((Service)this.containerLauncher).init(getConfig())
    > ((Service)this.containerLauncher).start()
```

所以,ContainerAllocator.EventType 类事件的接受者其实是 RMContainerAllocator。不过 RMContainerAllocator 并没有状态机,它的 handle()函数把这个事件挂在其内部的一个队列中,然后内部有个线程,从队列中取下这事件,最终通过其 handleEvent()函数处理接收到的事件。另外还要注意,虽然名曰 RMContainerAllocator,实际上却是在 NM 节点上,只相当于通往 RM 的中介而已。我们先看它的 handleEvent()函数:

```
[TaskAttemptImpl.RequestContainerTransition.transition()
=> RMContainerAllocator.handleEvent()]

RMContainerAllocator.handleEvent(ContainerAllocatorEvent event)
> if (event.getType() == ContainerAllocator.EventType.CONTAINER_REQ) {
>+ ContainerRequestEvent reqEvent = (ContainerRequestEvent) event
>+ JobId jobId = getJob().getID()
```

```

>+ Resource supportedMaxContainerCapability = getMaxContainerCapability() //最高限额
>+ if (reqEvent.getAttemptID().getTaskId().getTaskType().equals(TaskType.MAP)) {
    //这是个 Map 任务
>++ if (mapResourceRequest.equals(Resources.none())) {
>+++ mapResourceRequest = reqEvent.getCapability()
>+++ eventHandler.handle(new JobHistoryEvent(jobId,
    new NormalizedResourceEvent(org.apache.hadoop.mapreduce.TaskType.MAP,
        mapResourceRequest.getMemory())) //这个事件只是为历史记录
>+++ LOG.info("mapResourceRequest;" + mapResourceRequest)
>+++ if (mapResourceRequest.getMemory() > supportedMaxContainerCapability.getMemory()
    || mapResourceRequest.getVirtualCores() >
        supportedMaxContainerCapability.getVirtualCores()) {
    //所申请的内存或 VCore 超过了最高限额,无法满足
>++++ eventHandler.handle(new JobEvent(jobId, JobEventType.JOB_KILL))
>+++ } //end if
>+ } //end if (mapResourceRequest.equals(Resources.none()))
>+ reqEvent.getCapability().setMemory(mapResourceRequest.getMemory())
>+ reqEvent.getCapability().setVirtualCores(mapResourceRequest.getVirtualCores())
>+ scheduledRequests.addMap(reqEvent); //maps are immediately scheduled
    //进入 scheduledRequests 队列,等待被提交给 RM
    == RMContainerAllocator.ScheduledRequests.addMap(ContainerRequestEvent event)
>+> ...
>+> request = new ContainerRequest(event, PRIORITY_MAP)
>+>> this(event.getAttemptID(), event.getCapability(),
    event.getHosts(), event.getRacks(), priority)
>+> maps.put(event.getAttemptID(), request)
>+> addContainerReq(request)
>+ } else {
    //这是个 Reduce 任务
>+ if (reduceResourceRequest.equals(Resources.none())) {
>+++ ... //这部分与上列对 Map 任务的处理大致相同
>+ }
>+ reqEvent.getCapability().setMemory(reduceResourceRequest.getMemory())
>+ reqEvent.getCapability().setVirtualCores(reduceResourceRequest.getVirtualCores())
    //下面是对 Reduce 任务的特殊处理
>+ if (reqEvent.getEarlierAttemptFailed()) { //这是原先失败了的尝试
>+++ pendingReduces.addFirst(new ContainerRequest(reqEvent, PRIORITY_REDUCE))
    //add to the front of queue for fail fast,排在 pendingReduces 队列前面,以便尽快运行
>+ } else { //这是正常的 Reduce 任务尝试
>+++ pendingReduces.add(new ContainerRequest(reqEvent, PRIORITY_REDUCE))

```

```

//reduces are added to pending and are slowly ramped up,排在 pendingReduces 队列中
> ++ }
> + }
> } else if (event.getType() == ContainerAllocator.EventType.CONTAINER_DEALLOCATE) {
> + ...
> } else if (event.getType() == ContainerAllocator.EventType.CONTAINER_FAILED) {
> + ...
> }

```

我们在这里只关心对于 CONTAINER_REQ 的处理。虽然都是请求分配容器,这里对于 Map 任务与 Reduce 任务的处理有所不同,区别在于对 Reduce 任务有个特殊的处理,就是要为其创建一个 ContainerRequest 对象并把它排在一个 pendingReduces 队列中,而且对于失败重试的请求和原初的请求又有点不同。相比之下,对于 Map 任务就没有这样的考虑,而且 Map 任务的 ContainerRequestEvent 事件经过设置之后就被置入 scheduledRequests,等待被提交给 RM,而 Reduce 任务的请求就没有马上进入这个 List。这是因为:Map 任务应该尽早分配资源并投运;而 Reduce 任务则应暂缓申请。Hadoop 的 MapReduce 框架是个批处理式的工作流框架,并非 Mapper 一开始运行就会有数据流出,而是要到 Map 阶段完成运行时(还要考虑其排序操作)才能有数据流出。所以,让 Reducer 与 Mapper 同时获取资源并投运并无意义,Reducer 的资源申请可以延迟到 Mapper 接近完成时才提交给 RM。

另外,mapResourceRequest 和 reduceResourceRequest 都是 RMContainerAllocator 的内部成分,二者的类型均为 Resource,初始值都是 Resources.none()。

Map 任务的资源请求进入 scheduledRequests 这个队列后,实际上被转换成了一个 ContainerRequest,并被放入一个 maps 集合,等待被发送给 RM。

请注意这里 ContainerRequest 的创建。如前所述,每个任务,无论其为 MapTask 还是 ReduceTask,都需要有个容器,将来这个任务在哪个节点上运行就得向 RM 申请那个节点上的容器。所以,ContainerRequest 就是专门针对一个容器的申请,除所需资源 Capability 外,在这申请中可以指定节点机器 Hosts 和机架 Racks。这对于 MapTask 至关重要,因为一个 MapTask 的输入来自数据文件的一个特定的 Split,这个 Split 存储在哪里,这个 MapTask 就应该在哪里。HDFS 文件的数据块有多个复份,存储在不止一个的节点上,也可能在不止一个的机架上,所以 ContainerRequest 中给出的 Hosts 和 Racks 都是数组,让 RM 有了一定的灵活性。至于 ReduceTask,一般就无须指定节点和机架了,此时可以把 Hosts 和 Racks 设成 null。

那什么时候才会被提交给 RM 呢?

RMContainerAllocator 类是对抽象类 RMContainerRequestor 的扩充和落实,后者又是对 RMCommunicator 的扩充。所以 RMContainerAllocator 对象同时也是 RMCommunicator 对象。顾名思义,RMCommunicator 就是跟 RM 之间的通信员。RMCommunicator 对象内部有个线程 allocatorThread,这个线程通过心跳向 RM 提交资源申请。

```

RMCommunicator.allocatorThread.run()
> while (!stopped.get() && !Thread.currentThread().isInterrupted()) {
> + Thread.sleep(rmPollInterval)

```



```

>+ heartbeat()
>+ lastHeartbeatTime = context.getClock().getTime()
>+ executeHeartbeatCallbacks()
>+> while ((callback = heartbeatCallbacks.poll()) != null) {
>+>+ callback.run()
>+> }
> }

```

但是 heartbeat() 在 RMCommunicator 中是个抽象方法, 而 RMContainerAllocator 提供了这个方法:

```
[RMCommunicator.allocatorThread.run() > RMContainerAllocator.heartbeat()]
```

```
RMContainerAllocator.heartbeat()
```

```

> scheduleStats.updateAndLogIfChanged("Before Scheduling: ")
> List<Container> allocatedContainers = getResources()
>> Resource headRoom = getAvailableResources() == null?
    Resources.none() : Resources.clone(getAvailableResources())
>> AllocateResponse response = makeRemoteRequest() //向 RM 申请分配资源, 见后
>> Resource newHeadRoom = getAvailableResources() == null?
    Resources.none() : getAvailableResources()
>> List<Container> newContainers = response.getAllocatedContainers()
>> ... //不是我们此刻所关心, 略
>> return newContainers //返回后成为上面的 allocatedContainers
> if (allocatedContainers != null && allocatedContainers.size() > 0) {
>+ scheduledRequests.assign(allocatedContainers)
    == RMContainerAllocator.ScheduledRequests.assign(allocatedContainers)
> }
> int completedMaps = getJob().getCompletedMaps()
> int completedTasks = completedMaps + getJob().getCompletedReduces()
> if ((lastCompletedTasks != completedTasks) || (scheduledRequests.maps.size() > 0)) {
    //发生了变化
>+ lastCompletedTasks = completedTasks
>+ recalculateReduceSchedule = true
> }
> if (recalculateReduceSchedule) {
    //因为发生了变化, 就要对正在等待分配资源的 Reduce 任务进行调度
>+ preemptReducesIfNeeded()
>+ scheduleReduces(getJob().getTotalMaps(), completedMaps,
    scheduledRequests.maps.size(), scheduledRequests.reduces.size(),
    assignedRequests.maps.size(), assignedRequests.reduces.size(),

```

```

        mapResourceRequest, reduceResourceRequest,
        pendingReduces.size(), maxReduceRampupLimit, reduceSlowStart)
>+ recalculateReduceSchedule = false
> }
> scheduleStats.updateAndLogIfChanged("After Scheduling: ")

```

向 RM 申请资源,是由 RMContainerRequestor 通过对 RM 节点的 RPC 调用而完成的。

```

[RMCommunicator.allocatorThread.run() > RMContainerAllocator.heartbeat() >
getResources() > makeRemoteRequest()]

```

```

RMContainerRequestor.makeRemoteRequest()
> blacklistRequest = ResourceBlacklistRequest.newInstance(
    new ArrayList<String>(blacklistAdditions), new ArrayList<String>(blacklistRemovals))
> allocateRequest = AllocateRequest.newInstance(lastResponseID,
    super.getApplicationProgress(), new ArrayList<ResourceRequest>(ask),
    new ArrayList<ContainerId>(release), blacklistRequest) //构筑请求报文
> AllocateResponse allocateResponse = scheduler.allocate(allocateRequest)
    //这个 scheduler 就是向 RM 节点进行有关资源调度的 RPC 中介
    == ApplicationMasterProtocolPBClientImpl.allocate(allocateRequest)
>> r = proxy.allocate(null, requestProto) //proxy 就是 RM 节点的代理
>> return new AllocateResponsePBImpl(r)
> lastResponseID = allocateResponse.getResponseId()
> availableResources = allocateResponse.getAvailableResources() //RM 的回应中有资源清单
> lastClusterNmCount = clusterNmCount
> clusterNmCount = allocateResponse.getNumClusterNodes()
> ask.clear()
> release.clear()
> blacklistAdditions.clear()
> blacklistRemovals.clear()
> return allocateResponse

```

这里的 scheduler,就是向 RM 节点进行有关资源调度的 RPC 中介,它实际上是 ProtoBuf 向上层提供的 RPC 调用入口,所实现的界面是 ApplicationMasterProtocol。这个对象是在 RMCommunicator.serviceStart()中创建的:

```

RMCommunicator.serviceStart()
> RMCommunicator.scheduler = createSchedulerProxy()
>> scheduler = createSchedulerProxy()
>>> ClientRMProxy.createRMProxy(conf, ApplicationMasterProtocol.class)
>>>> createRMProxy(configuration, protocol, INSTANCE) == RMProxy.createRMProxy()
>>>>> ...
>>>>> RetryPolicy retryPolicy = createRetryPolicy(conf)

```

```

>>>>> InetAddress rmAddress = instance.getRMAddress(conf, protocol)
>>>>> T proxy = RMPProxy.<T>getProxy(conf, protocol, rmAddress)
>>>>> YarnRPC.create(conf).getProxy(protocol, rmAddress, conf)
>>>>> return (T) RetryProxy.create(protocol, proxy, retryPolicy)

```

但是这个所谓 scheduler 怎么又会在 RMContainerRequestor 和 RMContainerAllocator 中呢? 原来这二者其实都是 RMCommunicator, 因为这二者是对 RMCommunicator 直接或间接的扩充:

```

abstract class RMCommunicator extends AbstractService implements RMHeartbeatHandler {}
abstract class RMContainerRequestor extends RMCommunicator {}
class RMContainerAllocator extends RMContainerRequestor implements ContainerAllocator {}

```

所以, RMContainerRequestor 和 RMContainerAllocator 都是 RMCommunicator。既然这个 scheduler 是抽象类 RMCommunicator 的内部成分, 自然也就是 RMContainerRequestor 和 RMContainerAllocator 的内部成分。

经过 ProtocolBuffer 和 RPC 层, 这个 RPC 请求来到 RM 节点上, 转化成 RM 节点上的 ApplicationMasterProtocolPBServiceImpl.allocate():

```

ApplicationMasterProtocolPBServiceImpl.allocate(RpcController arg0,
                                                AllocateRequestProto proto)
> AllocateRequestPBImp request = new AllocateRequestPBImp(proto)
> AllocateResponse response = real.allocate(request)
> return ((AllocateResponsePBImp)response).getProto()

```

这里对 real.allocate() 的调用就是对 ApplicationMasterService.allocate() 的调用。RM 节点上的 ApplicationMasterService 就是专门为建立在各节点上的 AppMaster 服务的。各节点上 RMCommunicator 对 ApplicationMasterService 的心跳就相当于 NodeManager 对 RM 的心跳。

现在 ApplicationMasterService 必须通过其 allocate() 为来自 AM 的请求分配资源了。

```
[ApplicationMasterProtocolPBServiceImpl.allocate() > ApplicationMasterService.allocate()]
```

```

ApplicationMasterService.allocate()
> appAttemptId = amrmTokenIdentifier.getApplicationAttemptId()
> applicationId = appAttemptId.getApplicationId()
> this.amLivelinessMonitor.receivePing(appAttemptId) //这个 AM 还活着
> this.rmContext.getDispatcher().getEventHandler().handle(
    new RMAppAttemptStatusupdateEvent(appAttemptId, request.getProgress()))
    //对于具体 RMAppAttempt 的状态更新
> List<ResourceRequest> ask = request.getAskList() //要求分配的资源列表
> List<ContainerId> release = request.getReleaseList() //可以释放的资源列表
> ResourceBlacklistRequest blacklistRequest = request.getResourceBlacklistRequest()
    //对于黑名单的变更

```

```

> RMApp app = this.rmContext.getRMAApps().get(applicationId)
> ...
> Allocation allocation =
    this.rScheduler.allocate(appAttemptId, ask, release, blacklistAdditions, blacklistRemovals)
    == ResourceSchedulerWrapper.allocate(...) //分配资源
> appAttempt = app.getRMAppAttempt(appAttemptId)
> allocateResponse = recordFactory.newRecordInstance(AllocateResponse.class)
> ...
> allocateResponse.setAllocatedContainers(allocation.getContainers())
> ...
> return allocateResponse

```

从摘要中可以看出, `ApplicationMasterService.allocate()` 的作用不仅仅是分配资源, 实际上还起着处理心跳的作用。除容器分配以外, 这里也管状态的更新和容器的释放。

资源分配是通过 `rScheduler.allocate()` 完成的, 这是个 `ResourceSchedulerWrapper`, 但是最终还是要靠实际的 scheduler 解决问题。而这实际的 scheduler, 在我们这个情景中就是 `FifoScheduler`。

`FifoScheduler` 只是按先来后到进行调度, 这与前面为建立 AM 所需的资源分配没有多大不同。我们在前面看过 `FifoScheduler.allocate()` 的代码摘要, 这里就不重复了。反正, 只要还有资源可以分配, RM 总会发回一个响应, 里面载有所分配的容器。

于是我们这个情景的流程又转回 MRAppMaster 这一边。

```

[RMCommunicator.allocationThread.run() > RMContainerAllocator.heartbeat()
=> RMContainerAllocator.ScheduledRequests.assign()]

RMContainerAllocator.ScheduledRequests.assign(List<Container> allocatedContainers)
> Iterator<Container> it = allocatedContainers.iterator()
> LOG.info("Got allocated containers " + allocatedContainers.size())
> while (it.hasNext()) { //对于所分配的每一个容器
>+ Container allocated = it.next()
>+ Priority priority = allocated.getPriority()
>+ Resource allocatedResource = allocated.getResource()
>+ if (PRIORITY_FAST_FAIL_MAP.equals(priority) || PRIORITY_MAP.equals(priority)) {
>++ if (ResourceCalculatorUtils.computeAvailableContainers(allocatedResource,
    mapResourceRequest, getSchedulerResourceTypes()) <= 0 || maps.isEmpty()) {
>+++ isAssignable = false
>++ }
>+ } else if (PRIORITY_REDUCE.equals(priority)) {
>++ if (ResourceCalculatorUtils.computeAvailableContainers(allocatedResource,
    reduceResourceRequest, getSchedulerResourceTypes()) <= 0 || reduces.isEmpty()) {
>+++ isAssignable = false

```

```

>++ }
>+ } else {
>++ isAssignable = false
>+ }
>+ if(!isAssignable) {
>++ containerNotAssigned(allocated) //release container if we could not assign it
>++ it.remove()
>++ continue
>+ }
>+ String allocatedHost = allocated.getNodeId().getHost() //所分配容器指定的节点
>+ if(isNodeBlacklisted(allocatedHost)) { //如果这个节点在黑名单上就得要求替换
>++ ContainerRequest toBeReplacedReq = getContainerReqToReplace(allocated)
>++ if(toBeReplacedReq != null) {
>+++ ContainerRequest newReq = getFilteredContainerRequest(toBeReplacedReq)
>+++ decContainerReq(toBeReplacedReq)
>+++ if (toBeReplacedReq.attemptID.getTaskId().getTaskType() == TaskType.MAP) {
>++++ maps.put(newReq.attemptID, newReq)
>++++ } else {
>+++++ reduces.put(newReq.attemptID, newReq)
>++++ }
>+++ addContainerReq(newReq)
>++ }
>++ containerNotAssigned(allocated)
>++ it.remove()
>++ continue
>+ }
> } //end while
> assignContainers(allocatedContainers)
>> Iterator<Container> it = allocatedContainers.iterator()
>> while (it.hasNext()) { //将分配到的每一个容器指定给一个 TaskAttempt
>>+ Container allocated = it.next()
>>+ ContainerRequest assigned = assignWithoutLocality(allocated)
>>+ if(assigned != null) {
>>++ containerAssigned(allocated, assigned)
>>+> decContainerReq(assigned) //Update resource requests
>>+> e = new TaskAttemptContainerAssignedEvent(assigned.attemptID,
                                                allocated, applicationACLs)
>>+>> super(id, TaskAttemptEventType.TA_ASSIGNED) //向该 TaskAttempt 发出通知
>>+> eventHandler.handle(e)
>>+> assignedRequests.add(allocated, assigned.attemptID)

```

```

>> ++ it.remove()
>> ++ it.remove()
>> + }
>> }
>> assignMapsWithLocality(allocatedContainers)
> it = allocatedContainers.iterator()
> while (it.hasNext()) {
> + Container allocated = it.next()
> + LOG.info("Releasing unassigned and invalid container " + allocated
+ ". RM may have assignment issues")
> + containerNotAssigned(allocated)
> } //end while

```

从 RM 节点得到的容器都是用于 TaskAttempt 的。也就是说,分配到的容器是针对一项任务的一次尝试的,而不是针对一项任务的。现在,就把分配到的容器指定给第一批 TaskAttempt,并向它们发出通知 TA_ASSIGNED。而各个具体 TaskAttempt 的状态机对此事件的反应为:

```

addTransition(TaskAttemptStateInternal.UNASSIGNED, TaskAttemptStateInternal.ASSIGNED,
    TaskAttemptEventType.TA_ASSIGNED, new ContainerAssignedTransition())

```

即先执行 ContainerAssignedTransition.transition(),并转入 ASSIGNED 状态。

```

[RMCommunicator allocatorThread.run() > RMContainerAllocator.heartbeat()
=> ScheduledRequests.assign() > TaskAttemptEventType.TA_ASSIGNED
=> ContainerAssignedTransition.transition()]

```

ContainerAssignedTransition.transition(TaskAttemptImpl taskAttempt,

TaskAttemptEvent event)

```

> TaskAttemptContainerAssignedEvent cEvent = (TaskAttemptContainerAssignedEvent) event
> Container container = cEvent.getContainer() //从事件对象中抽取容器
> taskAttempt.container = container //这就是指定给这个 taskAttempt 的容器
> taskAttempt.remoteTask = taskAttempt.createRemoteTask()
    == MapTaskAttemptImpl.createRemoteTask() //假定其为 MapTask
>> MapTask mapTask = new MapTask("", TypeConverter.fromYarn(getID()), partition,
    splitInfo.getSplitIndex(), 1);
    //job file name is set in TaskAttempt, setting it null here
    // YARN doesn't have the concept of slots per task, set it as 1.
>> mapTask.setUser(conf.get(MRJobConfig.USER_NAME))
>> mapTask.setConf(conf)
>> return mapTask
> taskAttempt.jvmID = new WrappedJvmID(
    taskAttempt.remoteTask.getTaskID().getJobID(), //所属 Job 的 ID

```



```

        taskAttempt.remoteTask.isMapTask(),           //是否 MapTask
        taskAttempt.container.getId().getContainerId()) //容器的 ID
> taskAttempt.taskAttemptListener.registerPendingTask(taskAttempt.remoteTask,
                                                    taskAttempt.jvmID)
>> jvmIDToActiveAttemptMap.put(jvmID, task) //将此 Task 连同其 jvmID 记入一个 Map
        // A JVM not present in this map is an illegal task/JVM.
> taskAttempt.computeRackAndLocality() //计算 taskAttempt 的地域,例如 RACK_LOCAL
> ContainerLaunchContext launchContext = createContainerLaunchContext(
        cEvent.getApplicationACLs(), taskAttempt.conf, taskAttempt.jobToken,
        taskAttempt.remoteTask, taskAttempt.oldJobId, taskAttempt.jvmID,
        taskAttempt.taskAttemptListener, taskAttempt.credentials)
> e1 = new ContainerRemoteLaunchEvent(taskAttempt.attemptId,
        launchContext, container, taskAttempt.remoteTask)
>> super(taskAttemptID, allocatedContainer.getId(),
        StringInterner.weakIntern(allocatedContainer.getNodeId().toString()),
        allocatedContainer.getContainerToken(),
        ContainerLauncher.EventType.CONTAINER_REMOTE_LAUNCH)
> taskAttempt.eventHandler.handle(e1)
> e2 = new SpeculatorEvent(taskAttempt.getID().getTaskId(), -1)
>> super(Speculator.EventType.JOB_CREATE, timestamp)
> taskAttempt.eventHandler.handle(e2)

```

分配到容器之后,先通过 `createRemoteTask()` 创建一个“远程任务”`remoteTask`,在 MapReduce 框架中这只能是 `MapTask` 或 `ReduceTask`。注意,二者都是对 `Task` 类的扩充,不过这里仅仅是创建 `Task` 对象,并不意味着这个 `Task` 马上就会作为线程运行。

然后为其创建一个 `WrappedJvmID`。注意,这个所谓 `JvmID` 并非将来运行这个任务的 JVM 进程号,而只是相当于由三个元素构成的名称:JobID、是否 `MapTask`、和容器的 ID。

接着是为这个任务准备一个 CLC,即 `ContainerLaunchContext`,这是发起容器(实际上是容器所属的任务)运行所必需的,也直接决定了在启动任务时将使用的命令行。前面我们看到,在发起 `MRAppMaster` 的时候就有个 CLC,但是现在要的是针对具体任务的 CLC。这里通过 `createContainerLaunchContext()` 另外创建了一个 CLC,所需要的信息大多来自 `TaskAttemptImpl`,而 `TaskAttemptImpl` 中的信息则基本上来自原先创建 `MRAppMaster` 时的 CLC。有关这个针对具体任务尝试的 CLC 后面还要详述。

这样就万事俱备只欠投送和发起了,于是就产生一个 `ContainerRemoteLaunchEvent` 类的事件 `CONTAINER_REMOTE_LAUNCH`,用其驱动 `ContainerLauncher` 以完成远程任务的投送和发起。这个操作是异步的,不等其完成就会返回,所以又产生一个 `JOB_CREATE` 事件并用以驱动默认的 `Speculator`。所谓 `Speculator` 相当于“后备”、“替补”,运行中如果(推测)正式队员出了问题就让替补队员顶上去。不过我们在这里只关心前者,因为容器的投送和启动才是关键所在。

8.3 容器的跨节点投送和启动

作为“项目组长”，MRAppMaster 需要跨节点发起容器运行。比方说一个作业有 16 个 Mapper 和 1 个 Reducer，那么 MRAppMaster 就需要发起至少 17 个任务的运行，也就是要把至少 17 个容器投送到它们的目标节点上；这 17 个任务很可能不在同一个节点上，甚至有可能就是在 17 个不同的节点上。所以，容器的投送器 ContainerLauncher，实际上就是 ContainerLauncherImpl，即 MRAppMaster 的一个重要的成分。事实上，MRAppMaster 在其初始化阶段的 serviceStart() 中就创建了 ContainerLauncherImpl 对象，这个对象负责容器的投送和(相应任务的)发起。

而 ContainerLauncherImpl，则在其自己的 serviceStart() 中创建了一个事件处理线程 eventHandlingThread，并启动其运行。这个线程的 run() 函数是个 while 循环：

```
ContainerLauncherImpl.eventHandlingThread.run()
> while (!stopped.get() && !Thread.currentThread().isInterrupted()) {
>+ event = eventQueue.take()           //从 eventQueue 中摘下一个事件
>+ allNodes.add(event.getContainerMgrAddress())
>+ ...
>+ ep = createEventProcessor(event)     //为此事件创建一个 Runnable
>+> return new EventProcessor(event)
>+ launcherPool.execute(ep)           //从线程池中派发一个线程执行这个 Runnable
> }
```

这样，只要有事件进入 ContainerLauncherImpl 的事件队列，其 eventHandlingThread 就会为这个事件创建一个事实上的线程 EventProcessor，并执行其 run() 函数。注意这个 run() 函数中没有我们常见的 while 循环，而只是一次性地执行；所以，每创建一个 EventProcessor 并得到执行，就执行一次它的 run() 函数，对到来的事件进行处理。现在，到来的事件就是 CONTAINER_REMOTE_LAUNCH：

```
[ContainerAssignedTransition.transition() > EventType.CONTAINER_REMOTE_LAUNCH
=> ContainerLauncherImpl.EventProcessor.run()]
```

```
ContainerLauncherImpl.EventProcessor.run()
> ContainerId containerID = event.getContainerID()
> Container c = getContainer(event)
> switch(event.getType()) {
> case CONTAINER_REMOTE_LAUNCH:
>+ ContainerRemoteLaunchEvent launchEvent = (ContainerRemoteLaunchEvent) event
>+ c.launch(launchEvent) == Container.launch(launchEvent) //投送容器并发起其运行
>+> ContainerManagementProtocolProxyData proxy =
>+>+ getCMPProxy(containerMgrAddress, containerID)
>+> return cmProxy.getProxy(containerMgrBindAddr, containerID)
```

```

    == ContainerManagementProtocolProxy.getProxy() //获取通向目标节点的 proxy
>+> ContainerLaunchContext containerLaunchContext = event.getContainerLaunchContext()
>+> startRequest = StartContainerRequest.newInstance(
    containerLaunchContext, event.getContainerToken())
>+> List<StartContainerRequest> list = new ArrayList<StartContainerRequest>()
>+> list.add(startRequest) //把 StartContainerRequest 封装在一个 List 中
>+> StartContainersRequest requestList = StartContainersRequest.newInstance(list)
>+> StartContainersResponse response =
    proxy.getContainerManagementProtocol().startContainers(requestList)
    //通过 proxy 对目标节点进行 RPC 调用,返回 response
>+> if (response.getFailedRequests() != null &&
    response.getFailedRequests().containsKey(containerID)) {
>+>+ throw response.getFailedRequests().get(containerID).deserialize() //如果出错就发起异常
>+> }
>+> ByteBuffer portInfo = response.getAllServicesMetaData().get(
    ShuffleHandler.MAPREDUCE_SHUFFLE_SERVICEID)
>+> if (portInfo != null) port = ShuffleHandler.deserializeMetaData(portInfo)
>+> e = new TaskAttemptContainerLaunchedEvent(taskAttemptID, port)
>+>> super(id, TaskAttemptEventType.TA_CONTAINER_LAUNCHED)
>+> context.getEventHandler().handle(e) //将此事件发送给容器所属的 TaskAttemptImpl
>+> this.state = ContainerState.RUNNING // ContainerLauncherImpl 的状态进入 RUNNING
> case CONTAINER_REMOTE_CLEANUP:
>+ c.kill()
> }
> removeContainerIfDone(containerID)

```

因为到来的事件是 CONTAINER_REMOTE_LAUNCH, 这个 EventProcessor 线程就执行 Container.launch()。

可以说 Container.launch() 是整个过程的核心。因为这正是把任务投射到其目标节点并发起其在那里运行的关键一步。为此, 这里首先获取或创建通向目标节点上对应成分的代理, 再准备好一个 StartContainerRequest 对象, 然后就对这 proxy 进行 RPC 调用, 调用的函数就是 startContainers()。注意, 这里的调用参数并非(单数意义的)StartContainerRequest 对象, 而是一个(复数意义的)StartContainersRequest 对象, 实际上是多个启动容器请求的 List。所以哪怕实际上只是对单个容器的请求, 也要把它封装在一个 List 中, 再转换成一个 StartContainersRequest 对象。

这个节点上对于 proxy 的函数调用, 通过 ProtoBuf 和 RPC 层的作用, 会在目标节点上调用对应成分所提供的相同名称、相同参数表的函数:

```

[EventType.CONTAINER_REMOTE_LAUNCH
=> ContainerLauncherImpl.EventProcessor.run() > Container.launch()
=> ContainerManagementProtocolPBServiceImpl.startContainers()]

```

```
ContainerManagementProtocolPBServiceImpl.startContainers(RpcController arg0,  
                                                         StartContainersRequestProto proto)  
> request = new StartContainersRequestPBImp(proto)  
> StartContainersResponse response = real.startContainers(request)  
>> ContainerManagerImpl.startContainers(StartContainersRequest requests)  
> return ((StartContainersResponsePBImp)response).getProto()
```

这里的 `real` 是个 `ContainerManagerImpl` 对象,所以最终调用的是 `ContainerManagerImpl` 的 `startContainers()`。那以后的事是下一章的话题了,我们在这里只要知道在目标节点上会有这样的调用就可以了。注意,这里的目标节点并非 `ResourceManager` 节点,而是另一个 `NodeManager` 节点。

当目标节点上的程序从对于 `ContainerManagerImpl.startContainers()` 的调用返回时, 这边的 RPC 调用也就随之返回了, 返回值都在 `StartContainersResponse` 中。如果没有出错的话, 此时投送过去的容器其实是容器所属的任务, 已经在目标节点上发起运行了。于是就向这个容器 (和任务) 所属的 `TaskAttemptImpl` 对象发送一个 `TA_CONTAINER_LAUNCHED` 事件。

而 TaskAttemptImpl 对象的状态机对此事件的反应为:

```
addTransition(TaskAttemptStateInternal.ASSIGNED, TaskAttemptStateInternal.RUNNING,
    TaskAttemptEventType.TA_CONTAINER_LAUNCHED,
    new LaunchedContainerTransition())
```

先执行 `LaunchedContainerTransition.transition()`, 然后进入 `RUNNING` 状态:

```
LaunchedContainerTransition, transition(TaskAttemptImpl taskAttempt, TaskAttemptEvent evnt)
> event = (TaskAttemptContainerLaunchedEvent) evnt
> taskAttempt.launchTime = taskAttempt.clock.getTime() //启动时间
> taskAttempt.shufflePort = event.getShufflePort()
> taskAttempt.taskAttemptListener.registerLaunchedTask(
    taskAttempt.attemptId, taskAttempt.jvmID)
> InetAddress nodeHttpInetAddr =
    NetUtils.createSocketAddr(taskAttempt.container.getNodeHttpAddress())
> taskAttempt.trackerName = nodeHttpInetAddr.getHostName()
> taskAttempt.httpPort = nodeHttpInetAddr.getPort()
> taskAttempt.sendLaunchedEvents() //向 MRAppMaster 中的 JobImpl 发送事件
> e = new SpeculatorEvent(taskAttempt.attemptId, true, taskAttempt.clock.getTime())
>> super(Speculator.EventType.ATTEMPT_START, timestamp)
> taskAttempt.eventHandler.handle(e) //向 Speculator 发送事件
> taskAttempt.remoteTask = null
> e = new TaskTAttemptEvent(taskAttempt.attemptId,
    TaskEventType.T ATTEMPT_LAUNCHED)
```

```
> taskAttempt.eventHandler.handle(e) //向 TaskImpl 发送事件,用于统计目的
```

这里已经没有什么实质性的操作了,因为发起任务在目标节点上运行的目的已经达到,剩下的都是一些辅助性的操作了。最后给本次 TaskAttempt 发出的事件也反映了这一点: T_ATTEMPT_LAUNCHED 这个事件所引起的操作只是用于统计目的。

所以,实质性的操作还在目标节点那一边的 ContainerManagerImpl.startContainers()。

8.4 目标节点上的容器投运

我们还没有深入看目标 NM 节点上启动任务运行的过程。我们已经看到, RPC 和 ProtocolBuffer 层在目标节点上调用了当地 ContainerManagerImpl 的 startContainers(), 我们就从这里开始继续往下看。注意, CONTAINER_REMOTE_LAUNCH 这个事件发生在 MRAppMaster 所在的 NM 节点上,这个事件使得目标 NM 节点上 ContainerManagerImpl 对象的 startContainers() 方法得到执行。

```
[EventType.CONTAINER_REMOTE_LAUNCH => ContainerManagerImpl.startContainers()]

ContainerManagerImpl.startContainers(StartContainersRequest requests)
> UserGroupInformation remoteUgi = getRemoteUgi()
> NMTokenIdentifier nmTokenIdentifier = selectNMTokenIdentifier(remoteUgi)
> authorizeUser(remoteUgi, nmTokenIdentifier)
> for (StartContainerRequest request : requests.getStartContainerRequests()) {
    //对于同一请求中的每个容器
    >+ containerTokenIdentifier =
        BuilderUtils.newContainerTokenIdentifier(request.getContainerToken())
    >+ verifyAndGetContainerTokenIdentifier(request.getContainerToken(),
                                           containerTokenIdentifier)
    >+ containerId = containerTokenIdentifier.getContainerID()
    >+ startContainerInternal(nmTokenIdentifier, containerTokenIdentifier, request)
    >+> authorizeStartRequest(nmTokenIdentifier, containerTokenIdentifier)
    >+> if (containerTokenIdentifier.getRMIdentifier() != nodeStatusUpdater.getRMIdentifier()) {
    >+> }
    >+> updateNMTokenIdentifier(nmTokenIdentifier)
    >+> containerId = containerTokenIdentifier.getContainerID()
    >+> String containerIdStr = containerId.toString()
    >+> String user = containerTokenIdentifier.getApplicationSubmitter()
    >+> LOG.info("Start request for " + containerIdStr + " by user " + user)
    >+> ContainerLaunchContext launchContext = request.getContainerLaunchContext()
                                           //从请求中提取 CLC
    >+> Map<String, ByteBuffer> serviceData = getAuxServiceMetaData()
    >+> if (launchContext.getServiceData() != null && !launchContext.getServiceData().isEmpty()) {
```

```

>+> }
>+> Credentials credentials = parseCredentials(launchContext)
>+> Container container = new ContainerImpl(getConfig(), this.dispatcher,
context.getNMStateStore(), launchContext,
credentials, metrics, containerTokenIdentifier)
//在目标节点上重构 ContainerImpl
>+> ApplicationId applicationID = containerId.getApplicationAttemptId().getApplicationId()
>+> if (context.getContainers().putIfAbsent(containerId, container) != null) {
>+>+ throw RPCUtil.getRemoteException(
"Container " + containerIdStr + " already is running on this node!!")
>+> }
>+> if (!serviceStopped) {
>+>+ Application application =
new ApplicationImpl(dispatcher, user, applicationID, credentials, context)
//在目标节点上重构 ApplicationImpl
>+>+ if (null == context.getApplications().putIfAbsent(applicationID, application)) {
>+>+ LOG.info("Creating a new application reference for app " + applicationID)
>+>+ LogAggregationContext logAggregationContext =
containerTokenIdentifier.getLogAggregationContext()
>+>+ Map<ApplicationAccessType, String> appAcls =
container.getLaunchContext().getApplicationACLs()
>+>+ context.getNMStateStore().storeApplication(applicationID,
buildAppProto(applicationID, user, credentials, appAcls, logAggregationContext))
>+>+ e = new ApplicationInitEvent(applicationID, appAcls, logAggregationContext)
>+>+> super(appId, ApplicationEventType.INIT_APPLICATION)
//先是 ApplicationEventType.INIT_APPLICATION
>+>+ dispatcher.getEventHandler().handle(e) //将此事件发送给重构出来的 ApplicationImpl
>+>+ }
>+>+ this.context.getNMStateStore().storeContainer(containerId, request)
>+>+ e = new ApplicationContainerInitEvent(container)
>+>+> super(containerId, containerIdStr, ApplicationAttemptId(), ApplicationId(),
ApplicationEventType.INIT_CONTAINER)
//然后是 ApplicationEventType.INIT_CONTAINER
>+>+ this.container = container
>+>+ dispatcher.getEventHandler().handle(e) //在目标节点上向 Application 发送这个事件
>+>+ this.context.getContainerTokenSecretManager().startContainerSuccessful(
containerTokenIdentifier)
>+> } else {
>+>+ throw new YarnException("Container start failed as the NodeManager is "
+ "in the process of shutting down")
}

```



```

>+> }
    //从 startContainerInternal() 返回
>+ succeededContainers.add(containerId)
> } //end for
> return StartContainersResponse.newInstance( //返回的响应中包括成功和失败的容器数量
    getAuxServiceMetaData(), succeededContainers, failedContainers)

```

注意,现在我们在目标节点上。以前的那些 Container 和 Application 都是在 MRAppMaster 那一边,很可能是在另一个节点上,所以,现在要在目标节点上重建这些对象作为镜像,因为目标节点上也需要有这些对象,然后向重建出来的 Application,实际上是 ApplicationImpl 对象,先后发送两个事件。先是 ApplicationEventType.INIT_APPLICATION; 然后是 ApplicationEventType.INIT_CONTAINER。这两个事件都是在 startContainerInternal() 中发送的。这两个事件其实都是命令。ApplicationImpl 的状态机对前者的反应是:

```

addTransition(ApplicationState.NEW, ApplicationState.INITING,
    ApplicationEventType.INIT_APPLICATION, new AppInitTransition())

```

先执行 AppInitTransition.transition(), 然后从 NEW 跳变到 INITING 状态。

```

[ContainerManagerImpl.startContainers() > ApplicationEventType.INIT_APPLICATION
=> AppInitTransition.transition()]

```

```

AppInitTransition.transition(ApplicationImpl app, ApplicationEvent event)
> ApplicationInitEvent initEvent = (ApplicationInitEvent)event
> app.applicationACLs = initEvent.getApplicationACLs()
> app.aclsManager.addApplication(app.getAppId(), app.applicationACLs)
> app.logAggregationContext = initEvent.getLogAggregationContext()
> e = new LogHandlerAppStartedEvent(app.appId, app.user, app.credentials,
    ContainerLogsRetentionPolicy.ALL_CONTAINERS, app.applicationACLs,
    app.logAggregationContext))
>> super(LogHandlerEventType.APPLICATION_STARTED)
> app.dispatcher.getEventHandler().handle(e) //将事件发送给 LogAggregationService

```

除与“访问控制名单”ACL 有关的一点操作外,这一步发出了一个看似无关的 LogHandlerAppStartedEvent 事件,具体是 LogHandlerEventType.APPLICATION_STARTED。表面上这只与 Log, 即日志有关,可实际上这只是去 LogAggregationService 那儿弯了一下,真正的旅程还是在应用和容器初始化的轨道上。回想在 RM 节点上也是这样,先去 LogAggregationService 绕了一下之后才在 AppLogInitDoneTransition 中开启后续的流程。

之所以会有这样的安排,是因为在这整个过程中确实有许多方面的操作交汇在一起,但是又要保持模块化,否则就会乱成一团,成为“炒面式(Spaghetti)”代码。但是也只有两种办法:一种是保持一个总控程序作为主线,让它一会儿调用这个模块,一会儿调用那个模块;另一种就是让各个模块自己管理自己,客观的效果就是这样自然而然地互相串了起来。显然,

Hadoop 采用的是后者,这似乎更符合“面向对象”的精神和原则。

事实上,LogAggregationService 对 APPLICATION_STARTED 的处理会导致发出 APPLICATION_LOG_HANDLING_INITED 事件,这个事件反过来又引起 ApplicationImpl 状态机进一步的反应。

```
[ContainerManagerImpl.startContainers() > ApplicationEventType.INIT_APPLICATION
=> AppInitTransition.transition()>LogHandlerEventType.APPLICATION_STARTED
=> LogAggregationService.handle()]
```

```
LogAggregationService.handle(LogHandlerEvent event)
> switch (event.getType()) {
> case APPLICATION_STARTED:
>+ LogHandlerAppStartedEvent appStartEvent = (LogHandlerAppStartedEvent) event
>+ initApp(appStartEvent.getApplicationId(), appStartEvent.getUser(),
           appStartEvent.getCredentials(), appStartEvent.getLogRetentionPolicy(),
           appStartEvent.getApplicationAcls(), appStartEvent.getLogAggregationContext())
           //这只是日志服务中针对具体 App 的初始化
>+> verifyAndCreateRemoteLogDir(getConfig())
>+> initAppAggregator(appId, user, credentials, logRetentionPolicy,
                       appAcls, logAggregationContext)
>+> ApplicationEvent eventResponse = new ApplicationEvent(appId,
                       ApplicationEventType.APPLICATION_LOG_HANDLING_INITED)
>+> this.dispatcher.getEventHandler().handle(eventResponse) //将事件发送给 ApplicationImpl
>+ break
> case CONTAINER_FINISHED:
>+ ...
> case APPLICATION_FINISHED:
>+ ...
> }
```

而 ApplicationImpl 状态机对 APPLICATION_LOG_HANDLING_INITED 的反应则是:

```
addTransition(ApplicationState.INITING, ApplicationState.INITING,
              ApplicationEventType.APPLICATION_LOG_HANDLING_INITED,
              new AppLogInitDoneTransition())
```

即执行 AppLogInitDoneTransition.transition(),但维持 INITING 状态不变。

```
AppLogInitDoneTransition.transition(ApplicationImpl app, ApplicationEvent event)
```

```
> e = new ApplicationLocalizationEvent(
           LocalizationEventType.INIT_APPLICATION_RESOURCES, app))
> app.dispatcher.getEventHandler().handle(e) //向 ResourceLocalizationService 发送此事件
```

ApplicationImpl 向 ResourceLocalizationService 发出的这个事件,实质上是请求,就是

为这 App 的各个容器进行资源本地化。

我们在前一章中看到过由 ResourceLocalizationService.handle()着手处理资源本地化的过程,但那时候是在 AppMaster 所在的节点上为这个 AppMaster 进行资源本地化,而现在则是在属于这 App 的某个 Task 所在的节点之一上为这个 Task 进行资源本地化。前者是为 App 的管理,后者是为 App 的执行。对于一个具体的 App,AppMaster 只有一个,但属于这个 App 的 Task,即容器却可以有很多。而具体到一个特定的节点上,则属于同一个 App 的 Task,从而容器,也可以有好多个。

然后,就向(本节点上的)ApplicationImpl 发送一个 APPLICATION_INITED 事件。而 ApplicationImpl 的状态机对此的反应则是这样:

```
addTransition(ApplicationState.INITING, ApplicationState.RUNNING,
               ApplicationEventType.APPLICATION_INITED, new AppInitDoneTransition())
```

即先执行 AppInitDoneTransition.transition(), 然后从 INITING 状态跳变至 RUNNING 状态。

```
AppInitDoneTransition.transition(ApplicationImpl app, ApplicationEvent event)
> for (Container container : app.containers.values()) {
>+ e = new ContainerInitEvent(container.getContainerId())
>+> super(c, ContainerEventType.INIT_CONTAINER)
>+ app.dispatcher.getEventHandler().handle(e) //向本 App 的所有容器发送 INIT_CONTAINER
> }
```

如前所述,在一个节点上可能会被指派了属于同一 App 的多个任务,比方说 5 个 Mapper,因而就会有多个容器。所以,这里通过一个 for 语句向属于同一 App 的每个容器发出事件(实际上是命令)INIT_CONTAINER。

而每个容器的 ContainerImpl 对象则通过其状态机对此做出反应:

```
addTransition(ContainerState.NEW,
               EnumSet.of(ContainerState.LOCALIZING,
                           ContainerState.LOCALIZED,
                           ContainerState.LOCALIZATION_FAILED,
                           ContainerState.DONE),
               ContainerEventType.INIT_CONTAINER, new RequestResourcesTransition())
```

先执行 RequestResourcesTransition.transition(), 视执行结果转入不同的状态。我们在前一章中看过这个函数的代码摘要,它启动了具体容器的资源本地化过程。尽管需要加以本地化的资源内容和来源不同,本地化的流程却是一致的,所以不再重述。

值得注意的是,虽然向各容器发出了 INIT_CONTAINER,实际上是启动资源本地化的命令。资源的本地化是需要一些时间的。ApplicationImpl 状态机却很快便从 AppInitDoneTransition.transition()返回,并将其状态变成了 RUNNING。换言之,只要属于某个 App 的容器开始资源本地化,这个 App 就算已经处于 RUNNING 状态了。

到了一个容器完成其资源本地化时,这个容器的状态机就会转入 LOCALIZED 状态。

再回头看前面 `startContainerInternal()` 中向 `ApplicationImpl` 对象发出的第二个事件,那是 `INIT_CONTAINER`。注意,这是 `ApplicationEventType.INIT_CONTAINER`(前面的第一个事件为 `ApplicationEventType.INIT_APPLICATION`,它间接地导致了资源本地化)。显然,一个是对于 App 的初始化命令,一个是对于任务容器的初始化命令,但是二者的类型都是 `ApplicationEventType`,因而都是发送给 `ApplicationImpl` 的。`ApplicationImpl` 的状态机对 `INIT_CONTAINER` 事件的反应要看其处于什么状态。有关的跳变规则有三条:

```
addTransition(ApplicationState.NEW, ApplicationState.NEW,
               ApplicationEventType.INIT_CONTAINER, new InitContainerTransition())
addTransition(ApplicationState.INITING, ApplicationState.INITING,
               ApplicationEventType.INIT_CONTAINER, new InitContainerTransition())
addTransition(ApplicationState.RUNNING, ApplicationState.RUNNING,
               ApplicationEventType.INIT_CONTAINER, new InitContainerTransition())
```

这三条规则有个共同点,就是都执行 `InitContainerTransition.transition()`,而且都维持原有状态不变。

```
[ContainerManagerImpl.startContainers() > ApplicationEventType.INIT_CONTAINER
=> InitContainerTransition.transition()]
```

```
InitContainerTransition.transition(ApplicationImpl app, ApplicationEvent event)
> ApplicationContainerInitEvent initEvent = (ApplicationContainerInitEvent) event
> Container container = initEvent.getContainer()
> app.containers.put(container.getContainerId(), container)
> switch (app.getApplicationState()) {
> case RUNNING;
> + e = new ContainerInitEvent(container.getContainerId())
> +> super(c, ContainerEventType.INIT_CONTAINER) //形参 c 就是 ContainerId
> + app.dispatcher.getEventHandler().handle(e)
> case INITING; case NEW;
> + break
> }
```

这里的 `app.containers` 是目标节点上属于这同一个 App 的所有容器的集合,因为一个节点上可能会被安排执行属于同一个 App 的多个任务,从而就有着多个容器。

下面的 `switch` 语句告诉我们,当 `ApplicationImpl` 的状态机处于 `NEW` 或 `INITING` 状态时,对 `ApplicationEventType.INIT_CONTAINER` 事件是没有反应的,要等这个状态机进入了 `RUNNING` 状态之后才会有反应,这时候的反应就是向具体的容器发出 `ContainerEventType.INIT_CONTAINER` 事件。不过, `InitContainerTransition.transition()` 并非唯一会向具体容器发出这个事件的所在。我们刚才看到,当 `ApplicationImpl` 处于 `INITING` 状态而受到 `ApplicationEventType.APPLICATION_INITED` 事件的触发时,会执行 `AppInitDoneTransition.transition()`,在那里也会向容器,即 `ContainerImpl` 发出这种事件。

所以, `startContainerInternal()` 中向 `ApplicationImpl` 对象发出这种事件, 只是考虑到容器所属 App 的 `ApplicationImpl` 对象可能已经存在而采取的措施。

再回到前面那条主线上, 虽然我们无须再重复考察资源本地化的流程, 但是我们知道, 当容器完成了资源本地化以后会向其 `ContainerImpl` 对象发出一个 `RESOURCE_LOCALIZED` 事件, 而 `ContainerImpl` 对象的状态机会执行 `LocalizedTransition.transition()`, 并且在那里会调用 `sendLaunchEvent()`。我们在前一章中看过 `MRAppMaster` 的投运, 那时候要投运的是 `MRAppMaster.class`, 所以脚本中主要的命令行是“`…/bin/java …MRAppMaster…`”。但是现在的情况有所不同了。我们不妨从 `sendLaunchEvent()` 开始再大致过一遍:

```
ContainerImpl.sendLaunchEvent()
> ContainersLauncherEventType launcherEvent = DWContainersLauncherEventType.LAUNCH_CONTAINER
> if (recoveredStatus == RecoveredContainerStatus.LAUNCHED) {
>+ launcherEvent = ContainersLauncherEventType.RECOVER_CONTAINER
> }
> e = new ContainersLauncherEvent(this, launcherEvent)
>+ super(eventType)    //launcherEvent == LAUNCH_CONTAINER
> dispatcher.getEventHandler().handle(e) //将此事件发送给 ContainersLauncher
```

`ContainersLauncher` 是 `ContainerManagerImpl` 内部的一个成分, `ContainerManagerImpl` 在其构造函数中通过 `createContainersLauncher()` 创建这个对象。下面是这个类的摘要:

```
class ContainersLauncher extends AbstractService
    implements EventHandler<ContainersLauncherEvent> {}
] ContainerExecutor exec
] Dispatcher dispatcher
] ContainerManagerImpl containerManager
] ExecutorService containerLauncher = Executors.newCachedThreadPool(
    new ThreadFactoryBuilder().setNameFormat("ContainersLauncher # %d").build())
    //创建线程池及其管理者
] ContainersLauncher(Context context, Dispatcher dispatcher, ContainerExecutor exec,
    LocalDirsHandlerService dirsHandler, ContainerManagerImpl containerManager)
] handle(ContainersLauncherEvent event)
```

上面 `sendLaunchEvent()` 所调用的 `dispatcher.getEventHandler().handle()`, 其实就是 `ContainersLauncher.handle()`:

```
ContainersLauncher.handle(ContainersLauncherEvent event)
> Container container = event.getContainer()
> ContainerId containerId = container.getContainerId()
> switch (event.getType()) {
> case LAUNCH_CONTAINER:
>+ Application app =
    context.getApplications().get(containerId.getApplicationAttemptId().getApplicationId())
```

```

>+ ContainerLaunch launch = new ContainerLaunch(context, getConfig(), dispatcher,
        exec, app, event.getContainer(), dirsHandler, containerManager)
        //ContainerLaunch 是个 Callable
        //将其提交给线程池,安排线程予以运行

>+ containerLauncher.submit(launch)
>+ running.put(containerId, launch)
>+ break
> case RECOVER_CONTAINER:
>+ ...
>+ break
> }

```

对于 LAUNCH_CONTAINER 事件,这个函数创建了一个 ContainerLaunch 对象,然后通过 containerLauncher.submit()将其提交给线程池 containerLauncher。这里需要一些说明。ContainerLaunch 是 Callable,即对于 Callable 界面的实现,是供线程作为其程序主体使用的,Callable 的入口 call()就相当于 Runnable 的 run()。但是创建一个 Callable 并不意味着直接创建了一个线程,还得有线程来执行它才行,一般是把它提交给一个实现了 ExecutorService 界面的某类对象,那里面通常会有个线程池。而这里的 containerLauncher,见前面 ContainersLauncher 类的摘要,却并非实现了 ContainerLauncher 界面的某类对象(注意,ContainerLauncher 与 ContainersLauncher 不是同一个词),而不过是一个实现了 ExecutorService 界面的线程池管理者。所以,把 ContainerLaunch 提交给 containerLauncher,就早晚总会有个线程来加以执行,调用它的 call()函数。

我们已经不是第一次碰上 ContainerLaunch 以及 ContainerLaunch.call()了,读者可以回到前一章中看一下,这里就不重复介绍了。

那么这里的命令行是什么呢?那就是“/bin/java YarnChild …”。这个命令行将在目标节点上发起一个 JVM 进程,并让这个 Java 虚拟机执行 YarnChild.class。

8.5 Uber 模式下的本地容器分配与投运

在 Uber 模式,即“拼车”模式下,App 所需的容器由 LocalContainerAllocator 完成。

在 MRAppMaster 对象初始化阶段的 serviceStart()中,会创建一个 containerAllocator,负责向“中央”即 RM 节点索要资源。根据作业是否在 Uber 模式下执行,这个 containerAllocator 可以是个 LocalContainerAllocator 对象,也可以是个 RMContainerAllocator 对象。前面我们看了采用 RMContainerAllocator 时的情景,这是常规的模式,但是不妨也了解一下采用 Uber 模式时的情景。

我们看一下 LocalContainerAllocator 类的摘要,这是对 RMCommunicator 的扩充。

```

class LocalContainerAllocator extends RMCommunicator implements ContainerAllocator {}
] String nmHost
] int nmPort
] int nmHttpPort

```



```

] serviceInit(Configuration conf)
    > super.serviceInit(conf) == RMCommunicator.serviceInit(conf)
    > retryInterval =
        getConfig().getLong(MRJobConfig.MR_AM_TO_RM_WAIT_INTERVAL_MS, ...)
] heartbeat()
    > allocateRequest = AllocateRequest.newInstance(this.lastResponseID,
        super.getApplicationProgress(),
        new ArrayList<ResourceRequest>(), //本应是 ask, 现为空 List
        new ArrayList<ContainerId>(), //本应是 release, 现也为空 List
        null) //本应是 BlackList, 所以实际上并未向 RM 申请容器
    > scheduler.allocate(allocateRequest) //这是 RMCommunicator.scheduler, 一个 RMProxy
    > retrystartTime = System.currentTimeMillis()
] handle(ContainerAllocatorEvent event)

```

LocalContainerAllocator 实现了 ContainerAllocator 界面, 而后者又是对 EventHandler 的扩展, 所以 LocalContainerAllocator 也实现了 EventHandler 界面, 有个 handle() 函数。

回顾前面 RequestContainerTransition 的代码, TaskAttemptImpl 发出 CONTAINER_REQ 事件时, 在 Uber 模式下那里所调用的 EventHandler 就是 LocalContainerAllocator。

可是, 从 LocalContainerAllocator.heartbeat() 的摘要中可以看出, 每当通过心跳向 RM 发送资源请求时, 其 ask 表列都是空的, 表示不要求分配和指定容器。那么, 怎么满足 TaskAttemptImpl 的 CONTAINER_REQ 要求呢? 我们看一下它的 handle():

```

LocalContainerAllocator.handle(ContainerAllocatorEvent event)
    > if (event.getType() == ContainerAllocator.EventType.CONTAINER_REQ) {
    >+ ContainerId cID = ContainerId.newContainerId(getContext().getApplicationAttemptId(),
        this.containerId.getContainerId())
        //生成一个新的 ContainerId, this 就是 MRAppMaster 对象本身
        //this.containerId 就是用来运行 MRAppMaster 的那个容器的 ID
    >+ Container container = recordFactory.newRecordInstance(Container.class) //创建一个容器
    >+ container.setId(cID) //设置容器的 ID
    >+ NodeId nodeId = NodeId.newInstance(this.nmHost, this.nmPort) //生成本节点的 NodeId
    >+ container.setNodeId(nodeId) //设置该容器的地点为本节点
    >+ container.setContainerToken(null) //没有 Token
    >+ container.setNodeHttpAddress(this.nmHost + ":" + this.nmHttpPort) //本节点的 Web 地址
    >+ if(event.getAttemptID().getTaskId().getTaskType() == TaskType.MAP){ //对于 MAP 任务
    >++ jce = new JobCounterUpdateEvent(event.getAttemptID().getTaskId().getJobId())
    >++> super(jobId, JobEventType.JOB_COUNTER_UPDATE)
    >++ jce.addCounterUpdate(JobCounter.OTHER_LOCAL_MAPS, 1)
    >++ eventHandler.handle(jce) //向 JobImpl 发送 JOB_COUNTER_UPDATE 事件
    >+ } //end if
        //发送 TA_ASSIGNED 事件

```

```

>+ e = new TaskAttemptContainerAssignedEvent(event.getAttemptID(),
        container, applicationACLs) //这个 container 并非真的来自 RM
>+> super(id, TaskAttemptEventType.TA_ASSIGNED)
>+ eventHandler.handle(e) //向 TaskAttemptImpl 发送 TA_ASSIGNED
> } //end if

```

从这个 handle()函数的代码可见,当 LocalContainerAllocator 接到 CONTAINER_REQ 时,它并不是真的去向 RM 申请,而只是伪造一个虚假的容器,里面填写的信息就是这个 MRAppMaster 对象本身所在容器的信息。也就是说,它准备让这个作业的 Map 任务和 Reduce 任务与自己共享同一个容器。然后马上就向 TaskAttemptImpl 发送 TA_ASSIGNED 事件,谎称容器已经申请到了。

至于对 Map 任务的容器分配则有所不同,即先要向所属的 JobImpl 对象发送一个 JOB_COUNTER_UPDATE事件,事件中所携带的 OTHER_LOCAL_MAPS 计数为 1,这在 RMContainerAllocator.handleEvent()中也是一样的。

这样,不管是常规模式还是 Uber 模式,最后 TaskAttemptImpl 都会以为已经有了容器,只不过在 Uber 模式下这些容器“恰好”都在本地。

8.6 任务的启动

我们在本章的前面看到,TaskAttemptImpl 对 TA_ASSIGNED 事件的反应是先执行 ContainerAssignedTransition.transition(),然后从 UNASSIGNED 状态转入 ASSIGNED 状态:

```

addTransition(TaskAttemptStateInternal.UNASSIGNED, TaskAttemptStateInternal.ASSIGNED,
    TaskAttemptEventType.TA_ASSIGNED, new ContainerAssignedTransition())

```

这跟 TA_ASSIGNED 事件的来源无关。

我们也看到,在 ContainerAssignedTransition.transition()中会创建一个 remoteTask,然后通过 createContainerLaunchContext()另外创建一个专门针对具体任务尝试的 CLC,尽管原先创建 MRAppMaster 时有过一个由 RM 节点下达的 CLC。这是因为,每个具体的 TaskAttemptImpl 可能会被指派到不同的节点上,运行环境就可能有所不同。而且,更重要的是,每个具体 Task 的输入/输出也会不同,比方说 16 个 Mapper 的输入来自同一个文件,但是却来自这个文件中不同的块,这就各不相同了。

不过创建所需的信息大多来自 TaskAttemptImpl,而 TaskAttemptImpl 中的信息大多来自原先创建 MRAppMaster 时的 CLC。我们看一下 createContainerLaunchContext()的摘要:

```

[TaskAttemptImpl.ContainerAssignedTransition.transition() > createContainerLaunchContext()]

ContainerLaunchContext createContainerLaunchContext(...)
> commonContainerSpec = createCommonContainerLaunchContext( //CLC 的公共部分
    applicationACLs, conf, jobToken, oldJobId, credentials)
> Map<String, String> env = commonContainerSpec.getEnvironment()

```

```

> Map<String, String> myEnv = new HashMap<String, String>(env.size())
> myEnv.putAll(env)
> MapReduceChildJVM.setVMEnv(myEnv, remoteTask)
> List<String> commands = MapReduceChildJVM.getVMCommand(
                                taskAttemptListener.getAddress(), remoteTask, jvmID)
                                //从 MapReduceChildJVM 获取用来启动 JVM 的命令行
> myServiceData = new HashMap<String, ByteBuffer>()
> for (Entry<String, ByteBuffer> entry : commonContainerSpec.getServiceData().entrySet()){
>+ myServiceData.put(entry.getKey(), entry.getValue().duplicate())
> }
> ContainerLaunchContext container = ContainerLaunchContext.newInstance(
                                commonContainerSpec.getLocalResources(), myEnv, commands, ...)
> return container //注意这“container”是个 ContainerLaunchContext

```

这里我们关注的焦点在于:当这个容器被投运的时候,所用的 Shell 命令行是什么样的?显然这就是代码中的 commands,是由 MapReduceChildJVM 类的 getVMCommand()提供的。顾名思义,这就是启动 VM 即 Java 虚拟机的命令。这里需要说明,之所以固定使用由 MapReduceChildJVM 提供的命令行,是因为这是在 MRAppMaster 内部,这个事实决定了将要启动的必定是个用来实现 MapReduce 计算的 Java 类。

```

[TaskAttemptImpl.ContainerAssignedTransition.transition() > createContainerLaunchContext()
> MapReduceChildJVM.getVMCommand()]

```

```

MapReduceChildJVM.getVMCommand(
                                InetAddress taskAttemptListenerAddr, Task task, JVMId jvmID)
> TaskAttemptID attemptID = task.getTaskID() //attemptID 就是 TaskID
> JobConf conf = task.conf
> Vector<String> vargs = new Vector<String>(8)
>> vargs.add(MRApps.crossPlatformifyMREnv(task.conf, Environment.JAVA_HOME)
                                + "/bin/java")
> String javaOpts = getChildJavaOpts(conf, task.isMapTask())
> //将命令行变量 taskid 替换成实际的 attemptID
> javaOpts = javaOpts.replace("@taskid@", attemptID.toString())
> ...
> // Add main class and its arguments
> vargs.add(YarnChild.class.getName()); //Java 主类的名称是“YarnChild”
> vargs.add(taskAttemptListenerAddr.getAddress().getHostAddress()) //IP 地址
> vargs.add(Integer.toString(taskAttemptListenerAddr.getPort())) //端口号
> vargs.add(attemptID.toString()) // TaskAttemptID
> vargs.add(String.valueOf(jvmID.getId())) // jvmID,Java 虚拟机 ID
> vargs.add("1>" + getTaskLogFile(TaskLog.LogName.STDOUT)) //输入重定向

```

```
> vargs.add("2>" + getTaskLogFile(TaskLog.LogName.STDERR)) //输出重定向
> ...
```

从这段程序摘要中可以看出,最后形成的这个所谓 VMCommand,即用来启动 Java 虚拟机的 Shell 命令行是“.../bin/java... YarnChild...”。显然,这个命令行所启动的程序是目标节点上具体用户主目录下的/bin/java,这就是 Java 虚拟机 JVM;所装载执行的 Java 类则是 YarnChild。当然,这必定是个带有 main()方法的类。至于具体的 Mapper 或 Reducer,则间接由 YarnChild 投运。回顾当初 App 的投运,在目标节点上装载执行的则是 MRAppMaster。

在 ContainerAssignedTransition.transition()的代码中,TaskAttemptImpl 在创建了 CLC 之后接连发出两个事件。第一个其实是个命令,那就是发送给 ContainerLauncher 的 CONTAINER_REMOTE_LAUNCH,这是要发起容器的运行。另一个是发送给 Speculator 的 ATTEMPT_START,意在告知其有个 TaskAttempt 正在启动这么个事实,因为后面如果执行的情况不好,可能需要后备任务顶上。不过,我们在这里只关心“主旋律”,即容器的投运,而不关心 Speculator。

如前所述,ContainerLauncher 是个界面,创建 ContainerLauncher 时实际创建的是实现了这个界面的 ContainerLauncherRouter,它会根据是否运行于 Uber 模式而创建 LocalContainerLauncher 或 ContainerLauncherImpl。

对于常规的运行模式,ContainerLauncherImpl 会将 CLC 和容器投送到目标节点,让目标节点上的 ContainerManager 完成资源的本地化,这我们已经看到,然后就(在目标节点上)让宿主操作系统启动执行 CLC 中的命令行。

但是,如果是 Uber 模式,则情况大有不同,因为 Uber 模式就是“拼车”模式,是让最终要运行的任务(如 Mapper、Reducer)拼用 MRAppMaster 的那个容器,在 MRAppMaster 所在的 JVM 进程中运行。这里我们也要看一下 Uber 模式下的任务投运。

在 Uber 模式下,上述这个 CONTAINER_REMOTE_LAUNCH 事件的处理者是 LocalContainerLauncher。不过它的 handle()函数只是把作为参数传下的事件对象挂在它的队列 eventQueue 中,另有一个线程 EventHandler 负责实际的处理。这个线程的代码如下:

```
LocalContainerLauncher.EventHandler.run() {
    ContainerLauncherEvent event = null;
    // Collect locations of map outputs to give to reduces
    final Map<TaskAttemptID, MapOutputFile> localMapFiles =
        new HashMap<TaskAttemptID, MapOutputFile>();
    // _must_ either run subtasks sequentially or accept expense of new JVMs
    // (i.e., fork()), else will get weird failures when maps try to create/
    // write same dirname or filename: no chdir() in Java
    while (!Thread.currentThread().isInterrupted()) {
        try {
            event = eventQueue.take(); //从队列中摘下一个事件
        } catch (InterruptedException e) { // mostly via T_KILL?JOB_KILL?
            LOG.error("Returning, interrupted : " + e);
            break;
        }
    }
}
```

```

    }
    LOG.info("Processing the event " + event.toString());
    if (event.getType() == EventType.CONTAINER_REMOTE_LAUNCH) {
        final ContainerRemoteLaunchEvent launchEv =
            (ContainerRemoteLaunchEvent)event;
        // execute the task on a separate thread,另起一个线程并提交成为一个 Future
        Future<?> future = taskRunner.submit(new Runnable() {
            public void run() { //这个线程的 run 函数
                runTask(launchEv, localMapFiles); //这个新创的线程就做这么一件事
            }
        });
        // remember the current attempt
        futures.put(event.getTaskAttemptID(), future); //放在 futures 集合中
    } else if (event.getType() == EventType.CONTAINER_REMOTE_CLEANUP) {
        // cancel (and interrupt) the current running task associated with the event
        TaskAttemptId taId = event.getTaskAttemptID();
        Future<?> future = futures.remove(taId);
        if (future != null) {
            LOG.info("canceling the task attempt " + taId);
            future.cancel(true);
        }
        // send "cleaned" event to task attempt to move us from
        // SUCCESS_CONTAINER_CLEANUP to SUCCEEDED state (or
        // {FAIL|KILL}_CONTAINER_CLEANUP to {FAIL|KILL}_TASK_CLEANUP)
        context.getEventHandler().handle(new TaskAttemptEvent(taId,
            TaskAttemptEventType.TA_CONTAINER_CLEANED));
    } else {
        LOG.warn("Ignoring unexpected event " + event.toString());
    }
} //end while
}

```

这个线程反复循环,从事件队列 eventQueue 中获取事件,如果队列空就睡眠等待,若获取到事件就加以处理。

但是,对于事件 CONTAINER_REMOTE_LAUNCH,这里却忽略了命令中 REMOTE 的语义,将其改成了本地投运,那就是:为其单独创建一个新的 Runnable 线程,并将其提交给 taskRunner,那是一个 ExecutorService。这样,这个新创建的 Runnable 对象就会作为一个 Future 线程运行。同时,这里也把这个 Future 任务存放在集合 future 中,那是一个 Map<TaskAttemptId,Future<?>>,以后凭 TaskAttemptId 就可找到其 Future 线程。

这样,taskRunner 中的线程就会来调用这个 Runnable 的 run()函数,从而调用 runTask()。

```
[LocalContainerLauncher.EventHandler.run() > runTask()]
```

```
LocalContainerLauncher.EventHandler.runTask(ContainerRemoteLaunchEvent launchEv,
                                             Map<TaskAttemptID, MapOutputFile> localMapFiles)
> attemptID = launchEv.getTaskAttemptID()
> Job job = context.getAllJobs().get(attemptID.getTaskId().getJobId())
> int numMapTasks = job.getTotalMaps()           //从 Job 对象中获取 Map 任务的总数
> int numReduceTasks = job.getTotalReduces()     //从 Job 对象中获取 Ceduc 任务的总数
> org.apache.hadoop.mapreduce.v2.app.job.Task ytask = job.getTask(attemptID.getTaskId())
                                                    //YARN (tracking) Task
> org.apache.hadoop.mapred.Task remoteTask = launchEv.getRemoteTask()
                                                    //classic mapred Task
> e = new TaskAttemptContainerLaunchedEvent(attemptID, -1)
>> super(id, TaskAttemptEventType.TA_CONTAINER_LAUNCHED)
> context.getEventHandler().handle(e)
    //将此事件发送给相应的 TaskAttemptImpl 对象,使其转入 RUNNING 状态
> if (numMapTasks == 0) doneWithMaps = true      //如果本作业没有 Mapper
> if (remoteTask.isMapOrReduce()) {
>+ JobCounterUpdateEvent jce = new JobCounterUpdateEvent(
                                                    attemptID.getTaskId().getJobId())
>+> super(jobId, JobEventType.JOB_COUNTER_UPDATE)
>+ jce.addCounterUpdate(JobCounter.TOTAL_LAUNCHED_UBERTASKS, 1)
>+ if (remoteTask.isMapTask())
    jce.addCounterUpdate(JobCounter.NUM_UBER_SUBMAPS, 1)
>+ else
    jce.addCounterUpdate(JobCounter.NUM_UBER_SUBREDUCES, 1)
>+ context.getEventHandler().handle(jce) //将此事件发送给 JobImpl 对象,更新统计数字
> }
> runSubtask(remoteTask, ytask.getType(), attemptID,
              numMapTasks, (numReduceTasks > 0), localMapFiles)
> futures.remove(attemptID)
```

显然,实质性的操作在于 runSubtask(),其余都是辅助性的操作。

```
[LocalContainerLauncher.EventHandler.run() > runTask() > runSubtask()]
```

```
runSubtask(org.apache.hadoop.mapred.Task task, final TaskType taskType,
            TaskAttemptId attemptID, ...)
> org.apache.hadoop.mapred.TaskAttemptID classicAttemptID =
                                                    TypeConverter.fromYarn(attemptID)
> JobConf conf = new JobConf(getConfig())
```



```

> conf.set(JobContext.TASK_ID, task.getTaskID().toString())
> conf.set(JobContext.TASK_ATTEMPT_ID, classicAttemptID.toString())
> conf.setBoolean(JobContext.TASK_ISMAP, (taskType == TaskType.MAP))
> conf.setInt(JobContext.TASK_PARTITION, task.getPartition())
> conf.set(JobContext.ID, task.getJobID().toString())
> // Use the AM's local dir env to generate the intermediate step output files
> String[] localSysDirs = StringUtils.getTrimmedStrings(
    System.getenv(Environment.LOCAL_DIRS.name()))
> conf.setStrings(MRConfig.LOCAL_DIR, localSysDirs)
> LOG.info(MRConfig.LOCAL_DIR + " for uber task: " + conf.get(MRConfig.LOCAL_DIR))
> // mark this as an uberized subtask so it can set task counter
> conf.setBoolean("mapreduce.task.uberized", true)
> if (taskType == TaskType.MAP) { //投运 MapTask
>+ MapTask map = (MapTask)task
>+ map.setConf(conf)
>+ map.run(conf, umbilical) //调用 MapTask.run(),详见下述
>+ if (renameOutputs) {
>++ MapOutputFile renamed = renameMapOutputForReduce(conf, attemptID,
    map.getMapOutputFile())
>++ localMapFiles.put(classicAttemptID, renamed)
>+ }
>+ relocalize()
>+ File[] curLocalFiles = curDir.listFiles()
>+ for (int j = 0; j < curLocalFiles.length; ++j) {
>+> if(!localizedFiles.contains(curLocalFiles[j])) {
>+>+ boolean deleted = false
>+>+ if(curFC != null) deleted = curFC.delete(new Path(curLocalFiles[j].getName()), true)
    // this is recursive, unlike File delete()
>+>+ if(!deleted) LOG.warn("Unable to delete unexpected local file/dir " +
    curLocalFiles[j].getName() + ": insufficient permissions?")
>+> } //end if
>+ } //end for
>+ if( ++ finishedSubMaps == numMapTasks) doneWithMaps = true
    //MapTask 的投运已完成
> } else /* TaskType.REDUCE */ { //投运 ReduceTask
>+ conf.set(MRConfig.FRAMEWORK_NAME, MRConfig.LOCAL_FRAMEWORK_NAME)
>+ conf.set(MRConfig.MASTER_ADDRESS, "local"); // bypass shuffle
>+ ReduceTask reduce = (ReduceTask)task
>+ reduce.setLocalMapFiles(localMapFiles)
>+ reduce.setConf(conf)

```

```

>+ reduce.run(conf, umbilical)    //调用 ReduceTask.run(),详见下述
>+ relocate()
    //ReduceTask 的投运已完成
> }

```

到了这里, Uber 模式与常规模式就合流了, 因为 MapTask 和 ReduceTask 的执行流程与它们所在的地点无关。在常规的非 Uber 模式下, 这两种任务也要被执行, 只不过那是在目标节点上。下面我们就来考察这两种任务的运行。

8.7 MapTask 的运行

上面我们已经看到, 在指定运行 MapTask 的目标节点上, 其 LocalContainerLauncher 的 EventHandler 将辗转启动执行 MapTask 的 run() 函数, 我们再继续往下考察。

```
[LocalContainerLauncher.EventHandler.run() > runTask() > runSubtask() > MapTask.run()]
```

```
MapTask.run(final JobConf job, final TaskUmbilicalProtocol umbilical)
```

```

> this.umbilical = umbilical
> if (isMapTask()) {
>+ // If there are no reducers then there won't be any sort. Hence the map
>+ // phase will govern the entire attempt's progress.
>+ if (conf.getNumReduceTasks() == 0) {
>++ mapPhase = getProgress().addPhase("map", 1.0f)
>+ } else {
>++ // If there are reducers then the entire attempt's progress will be
>++ // split between the map phase (67%) and the sort phase (33%).
>++ mapPhase = getProgress().addPhase("map", 0.667f)
>++ sortPhase = getProgress().addPhase("sort", 0.333f)
>+ }
> }
> TaskReporter reporter = startReporter(umbilical)
> boolean useNewApi = job.getUseNewMapper()    //是新 API 还是老 API
> initialize(job, getJobID(), reporter, useNewApi) //MapTask 的初始化
> if (jobCleanup) { // check if it is a cleanupJobTask
>+ runJobCleanupTask(umbilical, reporter) //如果只是要求作业清扫,则清扫一下就返回
>+ return
> }
> if (jobSetup) {
>+ runJobSetupTask(umbilical, reporter) //如果只是为任务执行“打前站”,那准备一下就返回
>+ return
> }

```

```

> if (taskCleanup) {
>+ runTaskCleanupTask(umbilical, reporter) //如果只要求任务清扫,那清扫一下就返回
>+ return
> }
> if (useNewApi) { //如果是采用新 API 的 Mapper
>+ runNewMapper(job, splitMetaInfo, umbilical, reporter)
>+> taskContext =
        new org.apache.hadoop.mapreduce.task.TaskAttemptContextImpl(job,
                                                                    getTaskID(), reporter)
>+> org.apache.hadoop.mapreduce.Mapper<INKEY, INVALUE, OUTKEY, OUTVALUE>
        mapper = ReflectionUtils.newInstance(taskContext.getMapperClass(), job)
>+> org.apache.hadoop.mapreduce.InputFormat<INKEY, INVALUE> inputFormat =
        ReflectionUtils.newInstance(taskContext.getInputFormatClass(), job)
>+> // rebuild the input split
>+> split = getSplitDetails(new Path(splitIndex.getSplitLocation()), splitIndex.getStartOffset())
        //Mapper 的输入文件一般是分片的,每个 MapTask 一片,通常就是一个数据块
>+> LOG.info("Processing split: " + split)
>+> org.apache.hadoop.mapreduce.RecordReader<INKEY, INVALUE>
        input = new NewTrackingRecordReader<INKEY, INVALUE>(split,
                                                                    inputFormat, reporter, taskContext)
        //用于从输入文件读入
>+> job.setBoolean(JobContext.SKIP_RECORDS, isSkipping()) //碰到损坏的记录是否跳过
>+> if (job.getNumReduceTasks() == 0) { //如果不用 Reducer,则 Mapper 直接输出结果
>+>+ output = new NewDirectOutputCollector(taskContext, job, umbilical, reporter)
>+> } else {
>+>+ utput = new NewOutputCollector(taskContext, job, umbilical, reporter)
        //一般都要将 mapper 的输出收集起来供 reducer 处理
>+> }
>+> mapContext = new MapContextImpl<INKEY, INVALUE, OUTKEY, OUTVALUE>(
                                                                    job, getTaskID(), ...)
>+> mapperContext = new WrappedMapper<...>().getMapContext(mapContext)
>+> input.initialize(split, mapperContext) //输入流的初始化
>+> mapper.run(mapperContext) == Mapper.run() //开始执行用户指定的 Mapper
>+>> setup(context)
>+>> while (context.nextKeyValue()) {
>+>>+ map(context.getCurrentKey(), context.getCurrentValue(), context)
                                                                    //真正的 Map 操作
>+>> }
>+> mapPhase.complete()
>+> setPhase(TaskStatus.Phase.SORT)

```

```

>+> statusUpdate(umbilical)
>+> input.close()
>+> output.close(mapperContext)
> } else { //如果采用老 API 的 Mapper
>+ runOldMapper(job, splitMetaInfo, umbilical, reporter)
> }
> done(umbilical, reporter)

```

关于 MapTask 的运行过程,上面摘要中已经添加了注释,请读者自行阅读、理解。有些细节还将在后续章节中详述。

8.8 ReduceTask 的投运

与 MapTask 的投运过程相似,有些节点被指定运行 ReduceTask。

```
[LocalContainerLauncher.EventHandler.run().> runTask() > runSubtask() > ReduceTask.run()]
```

```

ReduceTask.run(JobConf job, final TaskUmbilicalProtocol umbilical)
> JobConf job, final TaskUmbilicalProtocol umbilical
> if (isMapOrReduce()) {
>+ copyPhase = getProgress().addPhase("copy")
>+ sortPhase = getProgress().addPhase("sort")
>+ reducePhase = getProgress().addPhase("reduce")
> }
> TaskReporter reporter = startReporter(umbilical)
> boolean useNewApi = job.getUseNewReducer()
> initialize(job, getJobID(), reporter, useNewApi)
> if (jobCleanup) {
>+ runJobCleanupTask(umbilical, reporter) //如果只是要求作业清扫,那清扫一下就返回
>+ return
> }
> if (jobSetup) {
>+ runJobSetupTask(umbilical, reporter) //如果只是为任务执行“打前站”,那准备一下就返回
>+ return
> }
> if (taskCleanup) {
>+ runTaskCleanupTask(umbilical, reporter) //如果只要求任务清扫,则清扫一下就返回
>+ return
> }
> codec = initCodec() //Initialize the codec,如果需要压缩
> RawKeyValueIterator rIter = null

```

```

> ShuffleConsumerPlugin shuffleConsumerPlugin = null
> Class combinerClass = conf.getCombinerClass()
> CombineOutputCollector combineCollector = (null != combinerClass)?
    new CombineOutputCollector(reduceCombineOutputCounter, reporter, conf) : null
> clazz = job.getClass(MRConfig.SHUFFLE_CONSUMER_PLUGIN,
    Shuffle.class, ShuffleConsumerPlugin.class)
    //以“org.apache.hadoop.mapreduce.task.reduce.Shuffle”为键,看 conf 中与否设置用于
    //shuffle 操作的插件,默认为 org.apache.hadoop.mapreduce.task.reduce.Shuffle
> shuffleConsumerPlugin = ReflectionUtils.newInstance(clazz, job) //创建插件对象
> LOG.info("Using ShuffleConsumerPlugin: " + shuffleConsumerPlugin)
> shuffleContext = new ShuffleConsumerPlugin.Context(getTaskID(),
    job, FileSystem.getLocal(job), ...)
> shuffleConsumerPlugin.init(shuffleContext)
> rIter = shuffleConsumerPlugin.run() == Shuffle.run() //将 MapTask 的输出复制过来
>> int eventsPerReducer = Math.max(MIN_EVENTS_TO_FETCH,
    MAX_RPC_OUTSTANDING_EVENTS / jobConf.getNumReduceTasks())
>> int maxEventsToFetch = Math.min(MAX_EVENTS_TO_FETCH, eventsPerReducer)
>> // Start the map - completion events fetcher thread
>> EventFetcher<K,V> eventFetcher = new EventFetcher<K,V>(reduceId,
    umbilical, scheduler, this, maxEventsToFetch)
>> eventFetcher.start() //用来接受 MapTask 是否已经完成运行的信息
>> // Start the map - output fetcher threads
>> boolean isLocal = localMapFiles != null
>> int numFetchers = isLocal?1 :
    jobConf.getInt(MRJobConfig.SHUFFLE_PARALLEL_COPIES, 5)
>> Fetcher<K,V>[] fetchers = new Fetcher[numFetchers]
>> if (isLocal) {
>>+ fetchers[0] = new LocalFetcher<K, V>(jobConf, reduceId,
    scheduler, merger, ... localMapFiles)
>>+ fetchers[0].start()
>> } else {
>>+ for (int i = 0; i < numFetchers; ++i) { //数量取决于 MapTask 的多少
>>++ fetchers[i] = new Fetcher<K,V>(jobConf, reduceId, scheduler, merger, ...) //数据复制者
>>++ fetchers[i].start() //启动其运行,开始从各个 MapTask 复制其输出的数据
>>+ }
>> }
>> // Wait for shuffle to complete successfully
>> while (!scheduler.waitUntilDone(PROGRESS_FREQUENCY)) {
>>+ reporter.progress()
>>+ if (throwable != null)

```

```

        throw new ShuffleError("error in shuffle in " + throwingThreadName, throwable)
    }
    >> eventFetcher.shutdown() //Stop the event - fetcher thread, 数据复制已经完成
    >> for (Fetcher<K,V> fetcher : fetchers) fetcher.shutdown() //Stop the map - output fetcher threads
    >> scheduler.close() //stop the scheduler
    >> copyPhase.complete(); // copy is already complete, 数据复制阶段到此结束
    >> taskStatus.setPhase(TaskStatus.Phase.SORT)
    >> reduceTask.statusUpdate(umbilical)
    >> // Finish the on - going merges...
    >> RawKeyValueIterator kvIter = null
    >> kvIter = merger.close()
    >> if (throwable != null)
        throw new ShuffleError("error in shuffle in " + throwingThreadName, throwable)
    > mapOutputFilesOnDisk.clear() // free up the data structures
    > sortPhase.complete() // sort is complete, 复制并排序的阶段已经结束
    > setPhase(TaskStatus.Phase.REDUCE) //开始进行 Reduce 计算
    > statusUpdate(umbilical)
    > Class keyClass = job.getMapOutputKeyClass()
    > Class valueClass = job.getMapOutputValueClass()
    > RawComparator comparator = job.getOutputValueGroupingComparator()
    > if (useNewApi) { //如果是采用新 API 的 Reducer
    >+ runNewReducer(job, umbilical, reporter, rIter, comparator, keyClass, valueClass)
    >+ RawKeyValueIterator rawIter = rIter
    >+ rIter = new RawKeyValueIterator()
    >+ taskContext = new org.apache.hadoop.mapreduce.task.TaskAttemptContextImpl(job,
        getTaskID(), reporter)
    >+ org.apache.hadoop.mapreduce.Reducer<INKEY, INVALUE, OUTKEY, OUTVALUE>
        reducer = ReflectionUtils.newInstance(taskContext.getReducerClass(), job)
    >+ org.apache.hadoop.mapreduce.RecordWriter<OUTKEY, OUTVALUE>
        trackedRW = new NewTrackingRecordWriter<OUTKEY, OUTVALUE>(this, taskContext)
    >+ job.setBoolean("mapred.skip.on", isSkipping())
    >+ job.setBoolean(JobContext.SKIP_RECORDS, isSkipping())
    >+ reducerContext = createReduceContext(reducer, job, getTaskID(),
        rIter, reduceInputKeyCounter, ...)
    >+ reducer.run(reducerContext) == Reducer.run(reducerContext) //reducer 开始执行
    >+> setup(context)
    >+> while (context.nextKey()) {
    >+>+ reduce(context.getCurrentKey(), context.getValues(), context) //真正的 Reduce 操作
    >+>+ // If a back up store is used, reset it
    >+>+ Iterator<VALUEIN> iter = context.getValues().iterator()

```



```

>+>>+ if(iter instanceof ReduceContext.ValueIterator) {
>+>>++ ((ReduceContext.ValueIterator<VALUEIN>)iter).resetBackupStore()
>+>>+ }
>+>> }
>+>> cleanup(context)
>+> trackedRW.close(reducerContext)
> } else { //如果是采用老 API 的 Reducer
>+ runOldReducer(job, umbilical, reporter, rIter, comparator, keyClass, valueClass)
> }
> shuffleConsumerPlugin.close()
> done(umbilical, reporter)

```

同样,摘要中已经添加了注释,有些细节还会在后续章节详述,但是读者不妨自己先看一下。比之 Mapper,Reducer 的复杂之处在于,在做 reduce()操作之前先要把 Mapper 的输入数据复制过来,而 Mapper 又通常不止一个。所以,这里需要有个称为 Shuffle 的过程来完成数据的复制和整理,在此过程中要有多个 Fetcher 并发进行数据的复制。

第 9 章

YARN 子系统的计算框架

Hadoop 中 YARN 子系统的使命是为用户提供大数据的计算框架。早期的 Hadoop,甚至早期的 YARN 都只提供一种计算框架,那就是 MapReduce。如前所述,MapReduce 是一种极简的,然而在很多情况下颇为有效的计算模型和框架。但是 Hadoop 的 MapReduce 框架要求使用者提供用 Java 语言编写的 Mapper 和 Reducer,而 App 本身则虽然简单但也要求用 Java 编写,这又使有些用户感到有点不便,而且 MapReduce 这个模式也过于简单和单调。所以 Hadoop 后来有了一些新的发展,除 MapReduce 外又提供了称为 Chain 和 Stream 的计算框架。一来使用户不必非得用 Java 编程;二来更允许用户利用 Linux 上的 Utility 工具软件搭建更像“数据流”的结构。本章介绍 YARN 子系统为用户提供的计算框架,当然主要还是传统的 MapReduce 框架。

9.1 MapReduce 框架

我们在前一章中看到,YARN 为一个 MapReduce 作业在某一 NM 节点上建立起类似于“项目组长”角色的 MRAppMaster;然后由它把这个作业分解成多个任务,再对每个任务进行尝试,为其在集群中的某个 NM 节点分配一个容器,并为其准备一个“容器投运上下文(Container Launch Context)”CLC,然后发起该容器在目标 NM 节点上投运。CLC 中包含着投运容器所需的各种信息,其中十分重要的就是对于宿主操作系统的命令行。在 Mapper 和 Reducer 容器的 CLC 里,这个命令行是“~/bin/java ... YarnChild ...”,那就是要启动一个 java 虚拟机,并让它执行 java 类 YarnChild。至于具体的 Mapper 或 Reducer,则是由 YarnChild 向 MRAppMaster 领取并加以本地化,然后投运。当然,不同 App 的 Mapper 和 Reducer 很可能是不同的。

其实,当初 MRAppMaster 自身也是这样投运的,只不过那时候由 ResourceManager 起着现在 MRAppMaster 所起的作用,而 CLC 中的命令行则是“.../bin/java... MRAppMaster...”。所以逻辑上这是两个不同层次的 CLC。

那么 YarnChild 起着什么作用呢?我们知道,由 JVM 装载运行的 Java 类必须带有“主函数”,即 main()函数,但是 MapTask 和 ReduceTask 都不是这样。因此就需要有个带 main()函数的 Java 类来起个开路、占位的作用,先把 JVM 启动起来,然后再由其负责装载启动作为任务的 Java 类。另外,目前 MapReduce 框架中只有 MapTask 和 ReduceTask 这样两种任务,最多还有 Combiner,但是将来也许还会有别的,我们需要有个统一的、稳定不变的格局,这就是所谓框架。所以,从这个意义上说,YarnChild 正是这个框架上的一个部位,它的作用是在

远程 NM 节点上作为一个 JVM 进程装载和运行普通不带 main() 函数的 Java 类, 例如用户提供的 Mapper 或 Reducer, 并与 MRAppMaster 保持联系。换句话说, 就是在项目组长的管理下扮演项目组成员的角色, 完成项目组长交下的任务。

下面是 YarnChild 的摘要, 我们从它的主函数 main() 开始。

```
class YarnChild {}
] main(String[] args)
> JobConf job = new JobConf(MRJobConfig.JOB_CONF_FILE)
> String host = args[0]           //命令行中带入 MRAppMaster 所在的主机名或 IP 地址
> int port = Integer.parseInt(args[1])           //命令行中带入 MRAppMaster 的端口号
> address = NetUtils.createSocketAddrForHost(host, port) //Socket 的 IP 地址和端口号
> TaskAttemptID firstTaskid = TaskAttemptID.forName(args[2])
> long jvmIdLong = Long.parseLong(args[3])
> JVMId jvmId = new JVMId(firstTaskid.getJobID(),
                        firstTaskid.getTaskType() == TaskType.MAP, jvmIdLong)
> // Create TaskUmbilicalProtocol as actual task owner.
> taskOwner = UserGroupInformation.createRemoteUser(firstTaskid.getJobID().toString())
> // 与 MRAppMaster 的关系就像婴孩与母亲的关系, 中间要有条脐带 umbilical
> TaskUmbilicalProtocol umbilical = taskOwner.doAs(
    new PrivilegedExceptionAction<TaskUmbilicalProtocol>() {}
] run()
> return RPC.getProxy(TaskUmbilicalProtocol.class,
    TaskUmbilicalProtocol.versionID, address, job)
//这个脐带其实就是一个 proxy
> // report non-pid to application master
> JvmContext context = new JvmContext(jvmId, "-1000")
> LOG.debug("PID: " + System.getenv().get("JVM_PID")) //获取 JVM 的进程号 pid
> JvmTask myTask = null
> // poll for new task
> for (int idle = 0; null == myTask; ++idle) {
>+ MILLISECONDS.sleep(sleepTimeMillisecs)
>+ myTask = umbilical.getTask(context) //从 MRAppMaster 认领任务, 这是个 JvmTask
>+ TaskUmbilicalProtocol.getTask(context)
//通过 Proxy 与 MRAppMaster 通信, 获取一个 JvmTask
> }
> task = myTask.getTask() //从 JvmTask 中还原出 Task, 是 MapTask 或 ReduceTask
> YarnChild.taskid = task.getTaskID()
> // Create the job-conf and set credentials
> configureTask(job, task, credentials, jt)
> // set job classloader if configured before invoking the task
> MRApps.setJobClassLoader(job) //MRApps 是个工具性的类
```

```

> // Create a final reference to the task for the doAs block
> Task taskFinal = task
> childUGI.doAs(new PrivilegedExceptionAction<Object>() {}
] run()
    > FileSystem.get(job).setWorkingDirectory(job.getWorkingDirectory()) //工作目录
    > taskFinal.run(job, umbilical); //run the task, MapTask 或 ReduceTask
> RPC.stopProxy(umbilical)
> DefaultMetricsSystem.shutdown()
> TaskLog.syncLogsShutdown(logSyncer)

```

在 MRAppMaster 那儿有个 TaskAttemptListenerImpl 对象, 是 MRAppMaster 在其 serviceInit() 中与 ContainerLauncher 等一起创建的, 相当于 MRAppMaster 中的一个职能部门。YarnChild 需要跟 MRAppMaster 所在节点上的 TaskAttemptListenerImpl 建立联系, 以维持与所属 MRAppMaster 的关系, 这种关系就像婴儿跟母亲之间的“脐带(umbilical)”一样。在这条 umbilical 上的通信规程是 TaskUmbilicalProtocol。当然, 这又是通过 RPC 和 PB 实现的。TaskAttemptListenerImpl 实现了这个规程, 时刻倾听来自那些由 MRAppMaster 所发起的 YarnChild 的声音。在具体 YarnChild 这一头则要创建一个通向 MRAppMaster 的代理 proxy, 这就成为 YarnChild 的 umbilical。这样, 以后按此规程对这个 umbilical 的函数调用就会转化成 MRAppMaster 所在节点上对其 TaskAttemptListenerImpl 的相应函数的调用。上面代码中的 umbilical.getTask() 就是这样:

```
[YarnChild.main() > umbilical.getTask() => TaskAttemptListenerImpl.getTask()]
```

```

TaskAttemptListenerImpl.getTask(JvmContext context) //应对来自 YarnChild 的认领请求
> JVMId jvmId = context.jvmId
> LOG.info("JVM with ID: " + jvmId + " asked for a task")
> WrappedJvmID wJvmID = new WrappedJvmID(jvmId.getJobId(), jvmId.isMap, jvmId.getId())
> // Try to look up the task. We remove it directly as we don't give multiple tasks to a JVM
> if (!jvmIDToActiveAttemptMap.containsKey(wJvmID)) { //这个 JVM 的 task 已经被领走了
>+ LOG.info("JVM with ID: " + jvmId + " is invalid and will be killed.")
>+ jvmTask = TASK_FOR_INVALID_JVM //返回一个空白的 JvmTask
> } else { //尚未被领走
>+ if (!launchedJVMs.contains(wJvmID)) { //这个 JVM 尚未登记
>++ jvmTask = null
>++ LOG.info("JVM with ID: " + jvmId +
    " asking for task before AM launch registered. Given null task")
>+ } else { //JVM 已登记, 任务尚未被领走
>++ org.apache.hadoop.mapred.Task task = jvmIDToActiveAttemptMap.remove(wJvmID)
    //从 jvmIDToActiveAttemptMap 集合中取出预先安排好的任务
>++ launchedJVMs.remove(wJvmID) //从 launchedJVMs 集合中撤掉已领取的任务
>++ LOG.info("JVM with ID: " + jvmId + " given task: " + task.getTaskID())

```

```

> ++ jvmTask = new JvmTask(task, false) //将此任务包装在一个 JvmTask 中
> + }
> }
> return jvmTask //返回该 JvmTask

```

TaskAttemptListenerImpl 内部维持着两个集合,一个是 launchedJVMs,另一个是 jvmIDToActiveAttemptMap。前者记录着下属所有已经登记为已投运但还没有领走任务的 JVM;后者则记录着所有尚未被领走的任务及其所属的 JVM。知道了这些,这段代码就容易理解了。最后被领走的 Task 需要被包装成 JvmTask,因为还需要附加一些辅助信息。

回到上面 YarnChild.main()的代码。从 MRAppMaster 领到了 JvmTask 之后,从中恢复出具体的 MapTask 或 ReduceTask,就可以调用其 run()方法,开始执行这个任务了。注意,Task 是个抽象类,具体就是 MapTask 或 ReduceTask。所以,这里的 taskFinal.run()实际调用的是 MapTask.run()或 ReduceTask.run()。

读者也许会问:为什么还要让 YarnChild 从 MRAppMaster 领取任务?不是有 CLC,还有资源本地化的过程吗?对此,Hadoop 的文档和代码注释中均未提及,我们只能分析猜测。

这里从 MRAppMaster 领取的任务是具体的 MapTask 或 ReduceTask 对象,而不只是任务说明。实际上 MapTask 和 ReduceTask 对象的体量往往还是不小的,且长度不一,放在 CLC 中总是不大合适。再说,如果同一节点上安排了 8 个 Map 任务,从而有 8 个 CLC,那样岂不是要将同一个 MapTask 传输 8 次?顺便说一下,现在这个安排也有这个问题。设想一下,假定一个节点上为同一个 MapTask 起了 8 个 JVM 进程,每个都要向 MRAppMaster 领取 MapTask 任务,那也同样有这个问题。但是,只要不是把 MapTask 本身包装在 CLC 中,要在这个问题上加以优化总还容易。而把任务的领取并入资源本地化,则确实值得考虑,不过其实通信流量也未必会有显著的不同。

既然上面代码中的 taskFinal.run()实际调用的是 MapTask 或 ReduceTask 的 run(),我们就来看这两种 Task 的 run()方法。

先看 MapTask,下面是这个类的摘要。

```

class MapTask extends Task {
] localizeConfiguration(JobConf conf)
] write(DataOutput out)
] readFields(DataInput in)
] run(final JobConf job, final TaskUmbilicalProtocol umbilical)
    > if (isMapTask()) {
    > + if (conf.getNumReduceTasks() == 0) mapPhase = getProgress().addPhase("map", 1.0f)
        //如果本作业中没有 ReduceTask,那 Mapper 的输出就不需要排序
        //所以 Map 操作的工作量就是这个任务的全部工作量
    > + else { //否则,如果有 Reducer,那就需要对输出进行排序:
    > ++ mapPhase = getProgress().addPhase("map", 0.667f) //Map 阶段的工作量算 2/3
    > ++ sortPhase = getProgress().addPhase("sort", 0.333f) //输出排序阶段的工作量算 1/3
    > + }
    > }

```

```

> initialize(job, getJobID(), reporter, useNewApi)
>> jobContext = new JobContextImpl(job, id, reporter)
>> taskContext = new TaskAttemptContextImpl(job, taskId, reporter)
>> if (getState() == TaskStatus.State.UNASSIGNED) setState(TaskStatus.State.RUNNING)
>> if (useNewApi) {
>>+ outputFormat = ReflectionUtils.newInstance(taskContext.getOutputFormatClass(), job)
//实际调用的是 JobContextImpl.getOutputFormatClass()
//如果未配置“mapreduce.job.outputformat.class”,即为 TextOutputFormat
>>+ committer = outputFormat.getOutputCommitter(taskContext)
== TextOutputFormat.getOutputCommitter(taskContext)
== FileOutputFormat.getOutputCommitter(taskContext)
>>+> new FileOutputCommitter(output, context)
//这个对象实现 Mapper 输出数据的格式化并将结果写入文件
>> } else {
>>+ committer = conf.getOutputCommitter()
>> }
>> Path outputPath = FileOutputFormat.getOutputPath(conf) //输出文件路径
>> if (outputPath != null) {
>>+ if ((committer instanceof FileOutputCommitter)) {
>>++ FileOutputFormat.setWorkOutputPath(conf,
((FileOutputCommitter)committer).getTaskAttemptPath(taskContext))
>>+ } else {
>>++ FileOutputFormat.setWorkOutputPath(conf, outputPath)
>>+ }
>> }
>> committer.setupTask(taskContext)
>> // ResourceCalculatorProcessTree 用于了解进程的资源使用情况
>> clazz = conf.getClass(MRConfig.RESOURCE_CALCULATOR_PROCESS_TREE,
null, ResourceCalculatorProcessTree.class)
//在 Linux 系统上是基于“/proc”的 ProcfsBasedProcessTree
>> pTree = ResourceCalculatorProcessTree.getResourceCalculatorProcessTree(
System.getenv().get("JVM_PID"), clazz, conf)
>> if (pTree != null) {
>>+ pTree.updateProcessTree()
>>+ initCpuCumulativeTime = pTree.getCumulativeCpuTime()
>> }
> if (useNewApi) runNewMapper(job, splitMetaInfo, umbilical, reporter) //采用新 API
> else runOldMapper(job, splitMetaInfo, umbilical, reporter) //采用老 API
> done(umbilical, reporter) //通知 MRAppMaster,本任务已经完成
] runNewMapper(JobConf job, TaskSplitIndex splitIndex,

```



```

        TaskUmbilicalProtocol umbilical, TaskReporter reporter)
    // 参数 splitIndex 表明这个 Map 任务的输入来自输入文件中的哪一个 Split
> taskContext = new TaskAttemptContextImpl(job, getTaskID(), reporter)
> mapClazz = taskContext.getMapperClass()
    //这就是用户给定的 Mapper 类,如果没有给定就采用默认的 Mapper.class
> mapper = ReflectionUtils.newInstance(mapClazz, job)
> inputFormat = ReflectionUtils.newInstance(taskContext.getInputFormatClass(), job)
> split = getSplitDetails(new Path(splitIndex.getSplitLocation()), splitIndex.getStartOffset())
    //根据 splitIndex 找到输入文件(是 HDFS 文件)中的某个块
    //HDFS 文件的每个块都是宿主文件系统中的文件
> input = new NewTrackingRecordReader<INKEY, INVALUE> (
        split, inputFormat, reporter, taskContext)
    //带 Tracking 功能的 RecordReader,这个对象将为 Mapper 提供输入
> if (job.getNumReduceTasks() == 0) {
>+ output = new NewDirectOutputCollector(taskContext, job, umbilical, reporter)
    //没有 Reducer,Mapper 的输出直接就是整个作业的输出
> } else {
>+ output = new NewOutputCollector(taskContext, job, umbilical, reporter)
    //有 Reducer,Mapper 的输出是供 Reducer 使用的
> }
> mapContext = new MapContextImpl<INKEY, INVALUE, OUTKEY, OUTVALUE>(
        job, getTaskID(), input, output, committer, reporter, split)
> mapperContext = new WrappedMapper<INKEY, ...>().getMapContext(mapContext)
>> return new Context(mapContext) // WrappedMapper.Context
> mapper.run(mapperContext) //执行 Mapper 的 run() 函数
> setPhase(TaskStatus.Phase.SORT) //Map 阶段结束,进入排序阶段
] getSplitDetails(Path file, long offset)
] createSortingCollector(JobConf job, TaskReporter reporter)
] runOldMapper(final JobConf job, ...)
    > runner = ReflectionUtils.newInstance(job.getMapRunnerClass(), job)
    > runner.run(in, new OldOutputCollector(collector, conf), reporter)
] class NewTrackingRecordReader<K,V> extends RecordReader<K,V> {}
] class TrackedRecordReader<K, V> implements RecordReader<K,V> {} //用于 OldMapper
] class SkippingRecordReader<K, V> extends TrackedRecordReader<K,V>{} //用于 OldMapper
] class NewDirectOutputCollector<K,V> extends RecordWriter<K,V> {}
] class NewOutputCollector<K,V> extends RecordWriter<K,V> {}
] class MapOutputBuffer<K extends Object, V extends Object> {}

```

关于老 API 和新 API,前面有过说明。这两种 API 所定义的 Mapper(和 Reducer)的类型是不一样的。提交作业时“作业单”中须作说明。这里就根据这个信息执行不同的 Mapper,分别执行 runNewMapper()或 runOldMapper()。对于 Reducer 也是如此。

MapTask 内嵌着一些用于输入/输出的内部类定义。其中 TrackedRecordReader 和 NewTrackingRecordReader 都是对 RecordReader 的扩充。RecordReader 就是从文件中按记录逐条读入。一个 KV 对,即键值对,就是一条记录。所谓 Tracked,或 Tracking,则是指在此基础上外加了一些跟踪统计之类的功能。而 SkippingRecordReader 又是对 TrackedRecordReader 的扩充,其特点是在读到已经损坏的记录时就设法跳过去继续往下读,而不是就此退出。NewDirectOutputCollector 和 NewOutputCollector 是对 RecordWriter 的扩充。前者用在 Reducer 缺席的情况下,此时 Mapper 的输出就是整个 MapReduce 计算的输出,后者用在有 Reducer 存在的情况下,此时 Mapper 的输出是要送给 Reducer 作为输入的,但是中间可能还要插入排序之类。

再看 MapTask 的 run() 函数,以及在 run() 里面调用的 runNewMapper()。代码本身并不复杂,又已加了注释,大体上应该不难理解。Hadoop 的 MapReduce 框架支持新老两种 API,不过事实上现在都已采用新 API,保留老 API 只是为了兼容,所以我们在这里只关心 runNewMapper()。

代码中的 taskContext 最终来自用户提供的 JobConf,通过 taskContext.getMapperClass() 得到的 mapper 就是当初用户通过 Job.setMapperClass() 提供的 Mapper。其他如输入格式等种种信息也来自 taskContext,但是最终也是来自用户的设置或默认。

Hadoop 的文件系统是分布式的 HDFS,一个大文件的内容纵向按固定大小分块存储在不同的节点上,称为 block。而每个具体 Mapper 的输入一般只是这个文件的一部分,称为一个“片”,即 split。一个片可以就是一个块,也可以是多个块,也可以只是一个块的一部分。具体 Mapper 的输入通常只是一个 split,所以要通过 getSplitDetails() 以 splitIndex 为索引获知具体 split 在文件中的位移等详情,而 splitIndex 来自 MRAppMaster,是在所领取的 MapTask 中设置好了的,可想而知其最终也是来自用户。

这里对通过 taskContext.getMapperClass() 获得用户提供的 Mapper 需要加些说明。代码中的 taskContext 是个 TaskAttemptContextImpl 类对象,这是对 JobContextImpl 的扩充。但是 TaskAttemptContextImpl 并未定义自己的 getMapperClass(),而 JobContextImpl 却定义了这个方法,所以 taskContext.getMapperClass() 就是 JobContextImpl.getMapperClass():

```
Class<? extends Mapper<?, ?, ?, ?>> getMapperClass() throws ClassNotFoundException {  
    return (Class<? extends Mapper<?, ?, ?, ?>>) conf.getClass(MAP_CLASS_ATTR, Mapper.class);  
}
```

这里的 conf 是个 JobConf 对象,这个类是对 Configuration 的扩充。MAP_CLASS_ATTR 则是 String“mapreduce.job.map.class”。这个方法的代码表明,其返回的结果是一个扩充了 Mapper 的某个类的 Class 对象(Java 的每个类都有一个用来描述其结构和属性的 Class 对象),这个 Class 对象是通过在 JobConf 中以“mapreduce.job.map.class”为键查找所得的结果。如果没有查到,就以 Mapper.class 为默认值。

用户提供的 Mapper 都是对 Mapper 类的扩充,并以用户提供的 map() 覆盖了 Mapper 类自身的 map(),但是一般不会去覆盖 Mapper 的 run() 方法。所以,代码中的 mapper.run() 就是 Mapper.run()。

知道了这些,我们再来看 Mapper 类的摘要,就清楚了:

```
[YarnChild.main() > doAs() > MapTask.run() > runNewMapper() > Mapper.run()]
```

```
class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {}
] abstract class Context
    implements MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {}
] setup(Context context)
] map(KEYIN key, VALUEIN value, Context context)
    > context.write((KEYOUT) key, (VALUEOUT) value)
] cleanup(Context context)
] run(Context context) // WrappedMapper.Context
    > setup(context)
    > while (context.nextKeyValue()) { //只要还有下一个 KV 对,就继续循环
    >+ map(context.getCurrentKey(), context.getCurrentValue(), context)
    > }
    > cleanup(context)
```

Mapper 的 `setup()` 和 `cleanup()` 其实都是空函数,是准备让用户在扩充 Mapper 时加以覆盖的,所以 `Mapper.run()` 中实际上就只有一个 `while` 循环。这个循环中只有一个语句,就是调用 `Mapper.map()`。不过,虽然只有一个语句,所涉及的操作其实也并不简单,因为调用 `map()` 时的参数需要由 `context.getCurrentKey()` 和 `context.getCurrentValue()` 提供,这就要涉及 Mapper 的输入机制,而 `map()` 内部则又会有 `context.write()`,这就涉及 Mapper 的输出机制。后面我们还要详述这个问题。

如上所述,用户一般都会提供自己扩充了的 Mapper,里面会以自己的 `map()` 来覆盖这里的 `map()` 函数。所以,这里在 `while` 循环中调用的一般都是用户提供的 `map()`。但是,如果用户没有提供自己的(扩充了的)Mapper,或者在自己的 Mapper 中没有提供自己的 `map()` 函数,那就会采用这个默认的 Mapper 而调用这里的 `map()` 了。这个 `map()` 貌似十分简单,就是每调用一次就把输入参数写出去,但是实际上这里往往隐含着类型和格式的转换,包括从 KEYIN 到 KEYOUT 和从 VALUEIN 到 VALUEOUT 的转换,而不只是简单地拷贝。

可见,作为一个线程,Mapper,或者对 Mapper 的扩充,起着类似于引擎的作用,是它在接连不断地调用着它的 `map()`。

不过 MapTask 其实比这里所见的更复杂,后面我们还要回头来看 MapTask。

最后还要说明,Mapper 虽然有个名为 `run()` 的方法,其自身却并非线程,甚至也不是 `Runnable` 或 `Callable`,这只是在 MapTask 的上下文中被调用的模块,MapTask 才是线程。

再看 ReduceTask,这比 MapTask 要复杂一些。我们在上一章中看过 `ReduceTask.run()` 的摘要,现在一方面重温一下,一方面也看看 ReduceTask 的全貌。

```
class ReduceTask extends Task {}
] setLocalMapFiles()
] initCodec() //if map-outputs are to be compressed,用于解压缩
] localizeConfiguration(JobConf conf)
] write(DataOutput out)
```

```

> super.write(out)
> out.writeInt(numMaps) // write the number of maps
] readFields(DataInput in)
> super.readFields(in)
> numMaps = in.readInt()
] getMapFiles(FileSystem fs)
] run(JobConf job, final TaskUmbilicalProtocol umbilical)
    > if (isMapOrReduce()) {
    >+ copyPhase = getProgress().addPhase("copy")
    >+ sortPhase = getProgress().addPhase("sort")
    >+ reducePhase = getProgress().addPhase("reduce")
    > }
    > TaskReporter reporter = startReporter(umbilical) //创建负责报告进度的线程
    > combinerClass = conf.getCombinerClass()
        //用户可以设置是否在 Mapper 与 Reducer 之间加一个 Combiner
    > combineCollector = (null != combinerClass)?
        new CombineOutputCollector(reduceCombineOutputCounter, reporter, conf) : null
    > clazz = job.getClass(MRConfig.SHUFFLE_CONSUMER_PLUGIN,
        Shuffle.class, ShuffleConsumerPlugin.class)
        //用户可以设置采用什么 shuffler,默认 Shuffle.class
    > shuffleConsumerPlugin = ReflectionUtils.newInstance(clazz, job) //创建 shuffler
    > LOG.info("Using ShuffleConsumerPlugin: " + shuffleConsumerPlugin)
    > rIter = shuffleConsumerPlugin.run() //这通常就是 Shuffle.run()
        //Shuffle 实现了 ShuffleConsumerPlugin 界面
    > sortPhase.complete() //当 shuffleConsumerPlugin 退出 run()时,sort 阶段也就结束了
    > setPhase(TaskStatus.Phase.REDUCE) //接着就是 reduce 阶段
    > keyClass = job.getMapOutputKeyClass() //与 Mapper 输出的 Key 相匹配
    > valueClass = job.getMapOutputValueClass() //与 Mapper 输出的 Value 相匹配
    > RawComparator comparator = job.getOutputValueGroupingComparator()
        //用于 Key 的比较运算
    > if (useNewApi) runNewReducer(job, umbilical, reporter, rIter,
        comparator, keyClass, valueClass)
    > else runOldReducer(job, umbilical, reporter, rIter, comparator, keyClass, valueClass)
] runOldReducer(JobConf job, ...)
] runNewReducer(JobConf job, ...) //见后

```

说是 ReduceTask,其实干的事情不仅仅是 Reduce。整个过程分成三个阶段,即拷贝(copy)、排序(sort)和归并(reduce)。ReduceTask 与 MapTask 不是一对一的关系,好几个 MapTask 的输出要由一个 ReduceTask 加以处理,所以首先要将各个具体 MapTask 的输出拷贝到 ReduceTask 所在的节点上。所以,拷贝的过程实际上是个收集的过程。然后是 sort。但是整个 sort 的过程其实是由 MapTask 与 ReduceTask 合力完成的,这里所说的只是

ReduceTask 这一边的 sort。我们在 ReduceTask 的代码中只看到 `sortPhase.complete()` 和 `TaskStatus.Phase.REDUCE`, 却没有看到这个 `sortPhase` 是什么时候开始。其实 MapTask 这一边的输出机制中也包含了排序, 所以 sort 阶段其实在 MapTask 中就开始了, 但那只是单个 Mapper 输出数据的排序。而在 ReduceTask 这一边, 则要将来自多个 MapTask 的数据合在一起进行排序。由于各个 MapTask 的输出是经过排序的, 这天然就是 MergeSort, 即合并排序。这整个拷贝和排序的过程就是被称为 shuffle 的过程。

至于 combiner, 则是一种“预归并”, 就是在 reduce 之前先小范围地 reduce 一下, 所以 Combiner 的代码常常就是 Reducer 的代码。

到了 Shuffle 操作完成, 这才真正开始 reduce 的操作。从这里我们也可以看出, Hadoop 的 MapReduce 是面向(大粒度的)批处理的, 而不是面向(小粒度的)流处理的。正是排序的存在使数据流变成了工作流。这个问题后面还要细讲。

下一步是 `runNewReducer()`。

```
[YarnChild.main() > doAs() > ReduceTask.run() > runNewReducer()]
```

```
ReduceTask.runNewReducer(JobConf job, ...)
```

```
> taskContext = TaskAttemptContextImpl(job, getTaskID(), reporter)
> rclazz = taskContext.getReducerClass()
// 获取用户给定的 Reducer 类名, 默认 Reducer.class
> reducer = ReflectionUtils.newInstance(rclazz, job) // 创建 Reducer 对象
> trackedRW = new NewTrackingRecordWriter<OUTKEY, OUTVALUE>(this, taskContext)
> reducerContext = createReduceContext(reducer, job, getTaskID(), ...)
> reducer.run(reducerContext)
```

这跟前面的 `runNewMapper()` 就很相像了, 也是从 `taskContext` 获取用户给定的 Reducer 类, 如果没有就默认 `Reducer.class`, 然后创建 Reducer 对象并调用其 `run()`。而且 Reducer 与 Mapper 的结构也很相像:

```
[YarnChild.main() > doAs() > ReduceTask.run() > runNewReducer() > Reducer.run()]
```

```
class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>{}
```

```
] abstract class Context
```

```
implements ReduceContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT>{}
```

```
] setup(Context context)
```

```
] reduce(KEYIN key, Iterable<VALUEIN> values, Context context)
```

// 如果用户在扩充 Reducer 时覆盖了这个 `reduce()`, 就不会调用它了

```
> for(VALUEIN value: values) {
```

```
> + context.write((KEYOUT) key, (VALUEOUT) value)
```

```
> }
```

```
] cleanup(Context context)
```

```
] run(Context context)
```

```
> setup(context)
```

```

> while (context.nextKey()) {
>+   reduce(context.getCurrentKey(), context.getValues(), context)
>+   Iterator<VALUEIN> iter = context.getValues().iterator()
>+   if (iter instanceof ReduceContext.ValueIterator) { //转入下一个 key 及其 value 序列
>++   ((ReduceContext.ValueIterator<VALUEIN>)iter).resetBackupStore()
>+   }
> }
> cleanup(context)

```

与 Mapper 相比,Reducer 的不同之处在于:Mapper 的输入是 KV 对,而 Reducer 的输入是一个 K 后面可以有多个 V,是个序列。例如,Mapper 的输入可能是这样:[研发部,张大力],[研发部,李小林],[财务部,王天亮],[人事部,赵明],等等;而在 Reducer 的输入端则是(假定按 key 排序):[财务部,王天亮],[人事部,赵明],[研发部,张大力,李小林],等等。正因为这样,map()函数的参数是 key 和 value,而 reduce()的参数是 key 和 values。于是,在调用 reduce()之前先得调用 context.getValues(),而 reduce()内部则会有个 for 循环。

刚才所举的例子很简单,但这并不意味着 KV 对都是这么简单的,事实上 V 可以很大、很复杂,例如这样的 KV 对也是可能的:“三国演义”,“……话说天下大势……”],[“红楼梦”,“第一回 甄士隐梦幻识通灵……”]。当然,相应的 Mapper 与 Reducer 也可能很复杂。

像 Mapper 一样,Reducer 虽然有个名为 run()的方法,其自身却并非线程,这是在 ReduceTask 的上下文中被调用的,ReduceTask 才是线程。

9.2 Streaming 框架

在 Hadoop 的典型应用中,Mapper 和 Reducer 都是 Java 类,是用 Java 语言编写的程序,在老版本的 Hadoop 中这是天经地义的。当然,在 Java 程序中也可以通过 JNI 连接到用 C 语言编写的程序,所以理论上用户也可以在 C 语言程序外面套上一层 Java 语言的壳,用来作为 Mapper 和 Reducer。但是,实际上很少有人这样用,对于一般用户而言那也太复杂了。然而,实际上这样的需求确实是存在的。而且,如前所述,所谓 Unix/Linux 的编程风格,以及由各种 shell 所提供的 Unix/Linux 程序设计环境,就是让用户可以用一种简单的数据流技术来灵活搭建相对比较复杂的应用和计算,这就是“管道(pipe)”和输入/输出重定向机制的本质。为此,Unix/Linux 在其/bin 和/usr/bin 等目录下提供了许多所谓 Utility,即小工具软件,例如 cat、wc、sort、uniq 等功能相对单一的小程序,用这些小程序搭成一维的链状数据流就可用来提供相当复杂的功能。所谓“流水线”、“流水作业”,也就是这个意思。而 MapReduce,就其结构和拓朴而言,其实也是一种数据流的机制,只不过它的这种数据流机制所形成的链长只是 2 (加上 Combiner 也只是 3),这方面它比 Unix/Linux 的 Pipe 机制还简单。但是,另一方面它将这个概念从单机扩展到了集群,概念上的单个计算节点(如 Mapper)实际上可以分布在很多物理节点上,这是它比 Unix/Linux 的 Pipe 机制复杂之处。但是,是否可以像 Unix 的 shell 一样,用那些现成的、用 C 语言编写的小工具软件来搭建 Hadoop 集群上的流水线呢?这显然是很有意义的。

正是出于这样的考虑,后来在 Hadoop 较新的版本中,特别是在 YARN 的设计中,就增加

了这样的机制,称为 Hadoop 的“流计算(streaming)”机制。这个所谓“流(stream)”,实际上就是“数据流(data flow)”。这样,在 Hadoop 集群上就可以通过例如下面这样的命令行启动一个数据流计算:

```
hadoop jar hadoop-streaming.jar -input myInputDirs -output myOutputDir \
    -mapper /bin/cat -reducer /usr/bin/wc
```

这在概念上相当于 Unix/Linux 上的命令行“cat myInputDirs | wc > myOutputDir”,但是有个重要的不同,那就是 Unix/Linux 命令行中的 cat 和 wc 都只是在单机上运行,哪怕是在多核的机器上也只有一份(一个进程),而 Hadoop 流中的 cat 却可以有很多份,作为 Mapper 分布在集群中,作为 Reducer 的 wc 也可以有多份。

上面命令行中 hadoop-streaming.jar 所封装的主类是 HadoopStreaming,这个类看上去似乎很简单:

```
class HadoopStreaming{ //invoked with the script "bin/hadoop jar hadoop-streaming.jar args"
    ] main(String[] args)
    ] printUsage()
```

显然这里只有 main()有实质的意义和作用,我们看它的代码:

```
[HadoopStreaming.main()]
```

```
public static void main(String[] args) throws Exception {
    if (args.length < 1) {
        System.err.println("No Arguments Given!");
        printUsage();
        System.exit(1);
    }

    int returnStatus = 0;
    String cmd = args[0];
    String[] remainingArgs = Arrays.copyOfRange(args, 1, args.length);
    if (cmd.equalsIgnoreCase("dumptb")) { //第一个命令行参数为“dumptb”
        DumpTypedBytes dumptb = new DumpTypedBytes();
        returnStatus = ToolRunner.run(dumptb, remainingArgs);
    } else if (cmd.equalsIgnoreCase("loadtb")) { //第一个命令行参数为“loadtb”
        LoadTypedBytes loadtb = new LoadTypedBytes();
        returnStatus = ToolRunner.run(loadtb, remainingArgs);
    } else if (cmd.equalsIgnoreCase("streamjob")) { //第一个命令参数为“streamjob”
        StreamJob job = new StreamJob();
        returnStatus = ToolRunner.run(job, remainingArgs);
    } else { // for backward compatibility
        StreamJob job = new StreamJob();
        returnStatus = ToolRunner.run(job, args); //运行中会调用 job.run()即 streamJob.run()
```

```

    }
    if (returnStatus != 0) {
        System.err.println("Streaming Command Failed!");
        System.exit(returnStatus);
    }
}

```

HadoopStream 命令行中的第一个选项其实相当于操作码,或者“命令”。这个选项可以是“loadtb”,其作用是“Load Typed Bytes”,即接受用键盘输入的<类型码,字节值>形式的内容,将其存储在文件中;也可以是与“loadtb”相反的“dumptb”,其作用是“Dump Typed Bytes”,即在终端上显示这样的文件内容。这二者其实都只是 MapReduce 示例程序,只不过所有的处理都集中在输入输出,既没有给定 Mapper 也没有给定 Reducer,所以都将采用默认的 Mapper 和 Reducer,这跟上述特定意义上的 stream 计算没有多大关系,也没有什么实用意义。但是,最后这个选项“streamjob”,或者干脆省略,就不一样了。这才是 HadoopStreaming 真正的用意所在,这时候后面的-mapper 和-reducer 选项才起作用,才能用一般(非 Java)的小工具软件搭建数据流,尽管也还只是很简单的数据流。所以,我们在这里只关心通过 ToolRunner 运行 StreamJob 类的情景。这个类的摘要是这样的:

```
[HadoopStreaming.main() > ToolRunner.run() > StreamJob.run()]
```

```

class StreamJob implements Tool {}
] CommandLineParser parser = new BasicParser()
] run(String[] args)
    > this.argv_ = Arrays.copyOf(args, args.length)
    > init()
    >> env_ = new Environment()
    > preprocessArgs()
    > parseArgv()
    >> cmdLine = parser.parse(allOptions, argv_)
    >> String[] values = cmdLine.getOptionValues("input")
    >> for (String input : values) inputSpecs_.add(input) //输入目录或文件路径
    >> output_ = cmdLine.getOptionValue("output") //输出目录
    >> mapCmd_ = cmdLine.getOptionValue("mapper") //用作 Mapper 的 Utility
    >> comCmd_ = cmdLine.getOptionValue("combiner") //用作 Combiner 的 Utility
    >> redCmd_ = cmdLine.getOptionValue("reducer") //用作 Reducer 的 Utility
    >> fsName = cmdLine.getOptionValue("dfs") //文件系统为 HDFS
    >> config_.set("fs.default.name", fsName)
    >> ...
    > postProcessArgs()
    > setJobConf() //设置好所用的 Mapper 和 Reducer 等
    > submitAndMonitorJob()

```

```

] go()
  > run(argv_)
] setJobConf()
  > jobConf_ = new JobConf(config_, StreamJob.class)
  > ...
  > jobConf_.set("stream.map.input", ioSpec_)
  > jobConf_.set("stream.map.output", ioSpec_)
  > jobConf_.set("stream.reduce.input", ioSpec_)
  > jobConf_.set("stream.reduce.output", ioSpec_)
  > isMapperACommand = false
  > c = StreamUtil.goodClassOrNull(jobConf_, mapCmd_, defaultPackage) //检查是否为 Java 类
  > if (c != null) jobConf_.setMapperClass(c) //mapper 是一个 Java 类
  > else { //mapper 并非 Java 类
  >+ isMapperACommand = true //mapper 是一个 shell 命令(utility)
  >+ jobConf_.setMapperClass(PipeMapper.class) //需要借用 Java 类 PipeMapper
  >+ jobConf_.setMapRunnerClass(PipeMapRunner.class) //以及 Java 类 PipeMapRunner
  >+ jobConf_.set("stream.map.streamprocessor", URLEncoder.encode(mapCmd_, "UTF-8"))
  > }
  > c = StreamUtil.goodClassOrNull(jobConf_, comCmd_, defaultPackage)
  > if (c != null) jobConf_.setCombinerClass(c) //combiner 是一个 Java 类
  > else {
  >+ jobConf_.setCombinerClass(PipeCombiner.class) //combiner 是一个 shell 命令(utility)
  >+ jobConf_.set("stream.combine.streamprocessor",
      URLEncoder.encode(comCmd_, "UTF-8"))
  > }
  > isReducerACommand = false
  > if (redCmd_ != null) { //命令行中指定了 reducer
  >+ if (redCmd_.compareToIgnoreCase("aggregate") == 0) { //并且是 Java 类 aggregate
  >++ jobConf_.setReducerClass(ValueAggregatorReducer.class)
  >++ jobConf_.setCombinerClass(ValueAggregatorCombiner.class)
  >+ } else { //指定了 reducer,但并非 aggregate
  >++ c = StreamUtil.goodClassOrNull(jobConf_, redCmd_, defaultPackage)
  >++ if (c != null) jobConf_.setReducerClass(c) // reducer 是一个 Java 类
  >++ else { //reducer 并非 Java 类
  >+++ isReducerACommand = true //reducer 是一个 shell 命令(utility)
  >+++ jobConf_.setReducerClass(PipeReducer.class) //需要借用 Java 类 PipeReducer
  >+++ jobConf_.set("stream.reduce.streamprocessor",
      URLEncoder.encode(redCmd_, "UTF-8"))
  >++ }
  >+ }

```

```

> } //end if (redCmd_!= null) - else
] submitAndMonitorJob()
  > jc_ = new JobClient(jobConf_)
  > running_ = jc_.submitJob(jobConf_) // JobClient.submitJob(jobConf_)
  > jobId_ = running_.getID()
  > jc_.monitorAndPrintJob(jobConf_, running_)
  > jc_.close()

```

除对命令行的解析外,run()中实质的操作就是 setJobConf()和 submitAndMonitorJob()。

从 setJobConf()可以看出,以 Mapper 为例,如果命令行中给定了 Mapper,那么首先要检查这是否为一个 Java 类,如果是就没有什么特别之处。如果不是,那就要把一个 Java 类 PipeMapper 设置成 Mapper。这是因为,Hadoop 的 MapReduce 框架是面向 Java 类的,非 Java 类的 Utility 装不到这个框架上去,必须有一个适配器在中间加以桥接,而 PipeMapper 就是这样的适配器。或者也可以说,PipeMapper 类对象的作用是包装 Linux 的 Utility 工具,使之可以被用作 Mapper。而且,相应地还得有一个 PipeMapRunner,用来取代 MapRunner。不过 Reducer(以及 Combiner)就要简单一些,只要有个 PipeReducer 就可以了。

再看作业的提交。显然,这里采用的是老 API,因为作业是通过 JobClient.submitJob()提交的。我们在前面看到,采用老 API 的作业是通过 JobClient.submitJob()提交的,而采用新 API 的作业则通过 Job.waitForCompletion()提交。不过新老 API 的作业提交流程最终会汇合到新 API 的 Job.submit(),新老 API 作业提交的流程从 Job.submit()以下就是一样的。但是,尽管两种 API 的作业提交流程汇合在一起,本作业是否采用新 API 的信息也一起被提交上去;当 MapTask 执行 Mapper 的时候又会分出新老,采用新 API 时会执行 runNewMapper(),而采用老API时会执行 runOldMapper();ReducerTask 也是如此。

上面说到 MapRunner,那就是在采用老 API 时才会用到。目前 Hadoop 仅在老 API 上支持 StreamJob。

所以,更重要的是 setJobConf()中对于所采用 Mapper 和 Reducer 的设置。如果所给定的 Mapper 不是一个 Java 类,那就要设置成 PipeMapper;而本应作为 Mapper 运行的 Utility 工具,即这里的 mapCmd_,则作为一项配置属性“stream.map.streamprocessor”添加在配置块 jobConf_中。同样,Combiner 就是 PipeCombiner,Reducer 就是 PipeReducer,而真正作为 Combiner 和 Reducer 运行的 comCmd_和 redCmd_,则作为配置项“stream.combine.streamprocessor”和“stream.reduce.streamprocessor”添加在 jobConf_中。

这样,以 Map 阶段为例,当作业内的一个 YarnChild 进程执行 MapTask.run()时,由于变量 useNewApi 的值为 false(来自 JobConf 中的属性“mapred.mapper.new-api”),就执行 runOldMapper():

```
[YarnChild.main()> doAs()> MapTask.run()> runOldMapper()]
```

```

runOldMapper(JobConf job, TaskSplitIndex splitIndex,
              TaskUmbilicalProtocol umbilical, TaskReporter reporter)
> InputSplit inputSplit = getSplitDetails(
    new Path(splitIndex.getSplitLocation()), splitIndex.getStartOffset())

```

```

//获取关于本任务输入 split 的信息,包括文件路径和起点
> updateJobWithSplit(job, inputSplit) //设置 inputSplit
> reporter.setInputSplit(inputSplit) //reporter 应报告这个 Split 的进度
> RecordReader<INKEY, INVALUE> in = isSkipping()?
    new SkippingRecordReader<INKEY, INVALUE>(umbilical, reporter, job) :
    new TrackedRecordReader<INKEY, INVALUE>(reporter, job)
//从数据源读入时如遇错误是否跳过错误继续往下进行
> job.setBoolean(JobContext.SKIP_RECORDS, isSkipping())
> numReduceTasks = conf.getNumReduceTasks() //从 JobConf 中获取 Reducer 数量
> if (numReduceTasks > 0) { //有 Reducer, Mapper 的输出应该收集后供 Reducer 使用
>+ collector = createSortingCollector(job, reporter)
> } else { //没有 Reducer, Mapper 的输出直接就是结果
>+ collector = new DirectMapOutputCollector<OUTKEY, OUTVALUE>()
>+ MapOutputCollector.Context context = new MapOutputCollector.Context(this, job, reporter)
>+ collector.init(context)
> }
> MapRunnable<INKEY, INVALUE, OUTKEY, OUTVALUE> runner =
    ReflectionUtils.newInstance(job.getMapRunnerClass(), job)
//从 JobConf 中获取 MapRunnable 的类型并加以创建,这次是 PipeMapRunner
//创建 PipeMapRunner 对象时会调用其 configure() 函数
> runner.run(in, new OldOutputCollector(collector, conf), reporter) == PipeMapRunner.run()
//执行 PipeMapRunner.run()
> mapPhase.complete()
> if (numReduceTasks > 0) setPhase(TaskStatus.Phase.SORT)
> statusUpdate(umbilical)
> collector.flush()

```

这里的 MapRunnable 是个界面,界面上只定义了一个函数,就是 run()。而 MapRunner 就是实现了这个界面的一个类,这个类与前面的 Mapper 很相似。这里的 runner 是通过反射机制的 newInstance() 创建的,具体创建的是何种对象取决于 job.getMapRunnerClass(),最终取决于提交作业时的设置。在我们这个情景中,如前面所见,这是 PipeMapRunner.class。

所以,这次执行的 MapRunner 是 PipeMapRunner。PipeMapRunner 是对 MapRunner 的扩充,提供了自己的 run() 函数。注意,通过 ReflectionUtils.newInstance() 创建某个类的对象时会调用其 configure() 函数(参见 Hadoop 中 ReflectionUtils 类的代码),但是 PipeMapRunner 并未提供自己的 configure() 函数,所以这里在创建 PipeMapRunner 对象的时候会调用 MapRunner.configure():

```

[MapTask.run() > runOldMapper() > ReflectionUtils.newInstance()
=> MapRunner.configure()]

```

```

MapRunner.configure(JobConf job)

```

```
> this.mapper = ReflectionUtils.newInstance(job.getMapperClass(), job) //这是 PipeMapper
>> PipeMapper.configure(job) //创建 PipeMapper 对象时又会调用其 configure()函数
> this.incrProcCount = SkipBadRecords.getMapperMaxSkipRecords(job)>0
    && SkipBadRecords.getAutoIncrMapperProcCount(job)
```

所以, PipeMapper 就相当于用户提供的那些 Mapper, 或者由系统提供的默认 Mapper。现在我们可以接着看 PipeMapRunner.run()了。

```
[YarnChild.main() > doAs() > MapTask.run() > runOldMapper() > PipeMapRunner.run()]
```

```
PipeMapRunner.run(RecordReader<K1, V1> input, OutputCollector<K2, V2> output,
    Reporter reporter)
> PipeMapper pipeMapper = (PipeMapper) getMapper() == MapRunner.getMapper()
>> return mapper //这是 PipeMapper
> pipeMapper.startOutputThreads(output, reporter);
> super.run(input, output, reporter) == MapRunner.run(input, output, reporter)
>> K1 key = input.createKey()
>> V1 value = input.createValue()
>> while (input.next(key, value)) {
>>+ mapper.map(key, value, output, reporter) == PipeMapper.map(key, value, output, reporter)
>> }
>> mapper.close()
```

显然, PipeMapRunner 仍会执行 MapRunner.run(), 不同之处是, 此前要进行一些准备, 主要就是 PipeMapper.startOutputThreads()。此外, 在执行 MapRunner.run()时所执行的 mapper.map()也成了 PipeMapper.map()。

PipeMapper 是对抽象类 PipeMapRed 的扩充, 我们先简单看一下 PipeMapper 的大致构成, 后面随着流程的进展再根据需要深入进去。

```
abstract class PipeMapRed {}
] ... //一些字段, 略
] String[] splitArgs(String args) //将命令行中的参数字符串分解成一个字符串数组
] configure(JobConf job) //这是关键所在
] startOutputThreads(OutputCollector output, Reporter reporter)
] createInputWriter(Class<? extends InputWriter> inputWriterClass)
] createOutputReader(Class<? extends OutputReader> outputReaderClass)
] class MROutputThread extends Thread {} //线程 MROutputThread 的定义
] class MRErrorThread extends Thread {} //线程 MRErrorThread 的定义
```

PipeMapRed 内部定义了两种线程, 即 MROutputThread 和 MRErrorThread, 这是干什么用的呢? 这就要从怎样把一个用作 Mapper 的 Utility 工具适配到 MapReduce 框架上的原理说起。姑且假定需要用作 Mapper 工具的是我们常用的 grep。我们知道, 像 grep 一类的工具都是作为独立的进程在运行的, 每个这样的进程都使用三个标准通道, 即 stdin、stdout 和

stderr, 除非程序中另外打开了文件或网络连接。而 Java 类 Mapper, 以及它的上一层 MapRunner, 则都是在线程 MapTask 中执行的。本来, 每当 MapRunner 通过 RecordReader 从输入 split 中读入了一个 KV 对, 就用它作为参数调用 mapper.map(), 而 mapper.map() 的输出则写入 MapOutputCollector, 例如 MapOutputBuffer。可是现在要用作 Mapper 的是工具软件 grep, 那么应该怎么办呢?

办法就是把 MapRunner 修改成一个特殊的 MapRunner, 例如 PipeMapRunner, 让它把读入的 KV 对通过操作系统的 pipe 机制写入 grep 的标准输入通道 stdin。但是 grep 不认 KV 对呀? 这倒好办, 只传一个 K 或 V 给它就行(当然要在 Reader 中作相应的安排)。而 grep 的输出, 则有两个通道, stdout 和 stderr, 我们可以在 MapTask 中创建两个线程, 一个是 MROutputThread, 另一个是 MRErrorThread, 让 grep 的 stdout 和 stderr 分别与 MROutputThread 和 MRErrorThread 的输入端对接, 再让 MROutputThread 把来自 grep 的数据写入 MapOutputCollector, 把来自 grep 的出错信息写入日志, 这就行了。于是, 这个 grep 进程就成了 MapReduce 框架上的一个外挂的 mapper。

进一步, 运用相同的原理, 还可以把这个外挂的 mapper 从一个进程推广到一个进程链, 例如“grep | wc -l”、“grep | tr [:upper:]'[:lower:]'”、“grep | sed”等。而且, 这个进程链中的“节点”也不一定是单个的 Utility 程序, 还可以是脚本。

Mapper 如此, Reducer 也是如此。

这样, 一方面 Linux 的那些 Utility 就可以在 Hadoop 集群上得到活用; 另一方面 Mapper 和 Reducer 的开发也不再局限于使用 Java 语言, 实际上可以采用在 Linux 环境中使用的任何语言。这确实是很诱人的解决方案。

明白了这个原理, 我们先回到上面 MapRunner.configure() 的代码中, 那里通过 job.getMapperClass() 获取用户设置的 MapperClass, 那当然就是 PipeMapper, 然后通过 ReflectionUtils.newInstance() 加以创建。只要是用 ReflectionUtils.newInstance() 创建某类对象, 如果这个类提供了 configure() 函数, 就会得到调用。所以, PipeMapper.configure() 就会得到调用:

```
[MapRunner.configure() > ReflectionUtils.newInstance() => PipeMapper.configure()]
```

```
PipeMapper.configure(JobConf job)
```

```
> super.configure(job) == PipeMapRed.configure(job)
> SkipBadRecords.setAutoIncrMapperProcCount(job, false)
> skipping = job.getBoolean(MRJobConfig.SKIP_RECORDS, false)
> if (mapInputWriterClass_.getCanonicalName().equals(
    TextInputWriter.class.getCanonicalName())) {
>+ String inputFormatClassName = job.getClass("mapred.input.format.class",
    TextInputFormat.class).getCanonicalName()
>+ ignoreKey = job.getBoolean("stream.map.input.ignoreKey",
    inputFormatClassName.equals(TextInputFormat.class.getCanonicalName()))
> }
> mapOutputFieldSeparator = job.get(
```

```

        "stream.map.output.field.separator", "\t").getBytes("UTF-8")
> mapInputFieldSeparator = job.get("stream.map.input.field.separator", "\t").getBytes("UTF-8")
> numOfMapOutputKeyFields = job.getInt("stream.num.map.output.key.fields", 1)

```

如前所述, PipeMapper 是对 PipeMapRed 的扩充, 所以这里的 super.configure() 就是 PipeMapRed.configure()。这一步很关键, 相比之下这里其他的那些设置就不那么重要, 所以留给读者自己慢慢看, 这里我们直接切入 PipeMapRed.configure()。

```

[MapRunner.configure() > ReflectionUtils.newInstance() => PipeMapper.configure()
> PipeMapRed.configure()]

```

PipeMapRed.configure(JobConf job)

```

> String argv = getPipeCommand(job)    //用作外挂 mapper 的那部分命令行
> joinDelay_ = job.getLong("stream.joindelay.milli", 0)
> job_ = job
> mapInputWriterClass_ = job_.getClass("stream.map.input.writer.class",
                                         TextInputWriter.class, InputWriter.class)
                                         //App 中或者配置文件中也许设置了这个
                                         //默认 TextInputWriter, 必须实现 InputWriter 界面
> mapOutputReaderClass_ = job_.getClass("stream.map.output.reader.class",
                                         TextOutputReader.class, OutputReader.class)
                                         //没有设置就默认 TextOutputReader
> reduceInputWriterClass_ = job_.getClass("stream.reduce.input.writer.class",
                                         TextInputWriter.class, InputWriter.class)
> reduceOutputReaderClass_ = job_.getClass("stream.reduce.output.reader.class",
                                         TextOutputReader.class, OutputReader.class)
> nonZeroExitIsFailure_ = job_.getBoolean("stream.non.zero.exit.is.failure", true)
> doPipe_ = getDoPipe()                //是否允许使用 pipe 机制, 在 PipeMapper 中固定返回 true
> if (!doPipe_) return                  //如果不允许就不往下走了
> setStreamJobDetails(job)
> String[] argvSplit = splitArgs(argv)  //把用作外挂 mapper 的那部分命令行转换成数组
> String prog = argvSplit[0]            //其中第一项就是程序名, 例如“grep”
> File currentDir = new File(".").getAbsoluteFile() //获取当前所在目录路径
> if (new File(prog).isAbsolute()) {     //如果可执行程序的路径是绝对路径
>+ // we don't own it. Hope it is executable
> } else {                               //可执行程序的路径是相对路径, 在当前目录之下
>+ FileUtil.chmod(new File(currentDir, prog).toString(), "a+x") //修改文件模式为可执行
> }
> if (!new File(argvSplit[0]).isAbsolute()) { //设置运行环境
>+ PathFinder finder = new PathFinder("PATH")
>+ finder.prependPathComponent(currentDir.toString())

```

```

>+ File f = finder.getAbsolutePath(argvSplit[0])
>+ if (f != null) {
>++ argvSplit[0] = f.getAbsolutePath()
>+ }
>+ f = null
> }
> LOG.info("PipeMapRed exec " + Arrays.asList(argvSplit))
> Environment childEnv = (Environment) StreamUtil.env().clone()
> addJobConfToEnvironment(job_, childEnv)
> addEnvironment(childEnv, job_.get("stream.addenvironment"))
> envPut(childEnv, "TMPDIR", System.getProperty("java.io.tmpdir"))
- - - - -
> ProcessBuilder builder = new ProcessBuilder(argvSplit) //构筑宿主操作系统上的命令行
> builder.environment().putAll(childEnv.toMap())
> Process sim = builder.start() == ProcessBuilder.start()
    //Start the process,执行该命令行,启动 mapper 进程,返回一个 Process 对象 sim
> strout = sim.getOutputStream() //获取该进程基于 stdout 通道的输出流
> clientOut_ = new DataOutputStream(new BufferedOutputStream(strout, BUFFER_SIZE))
    //在此基础上进一步构筑带缓冲的数据输出流
> strin = sim.getInputStream() //获取该进程基于 stdin 通道的输入流
> clientIn_ = new DataInputStream(new BufferedInputStream(strin, BUFFER_SIZE))
    //在此基础上进一步构筑带缓冲的数据输入流
> strerr = sim.getErrorStream() //获取该进程基于 stderr 通道的出错信息流
> clientErr_ = new DataInputStream(new BufferedInputStream(strerr))
    //在此基础上进一步构筑带缓冲的数据输入流
> startTime_ = System.currentTimeMillis()

```

这里的 `ProcessBuilder` 是由 JDK 提供的一个类,用来创建宿主操作系统上的进程。创建 `ProcessBuilder` 对象时的参数 `argvSplit` 是一个数组,其内容在创建进程时将被用作命令行,而 `ProcessBuilder.start()` 则创建这个进程。对于这个 `MapTask` 所在的 JVM 而言,所创建的进程 `sim` 就是其子进程,是个 `Process` 对象,所以在父进程中可以通过 `getInputStream()`、`getOutputStream()` 和 `getErrorStream()` 获取其三个标准输入输出通道。

按理说像创建外挂 mapper 子进程这样的动作不应该躲在 `configure()` 这样不太引人注目的函数里,这可是个很重要的大动作。

但是不管怎么说,在创建 `PipeMapper` 对象时它的 `configure()` 函数受到调用,这又导致 `PipeMapRed.configure()` 受到调用,于是就创建了外挂的 mapper 进程,在我们这个例子中就是一个 `grep` 进程。

然后就是 `PipeMapper.startOutputThreads()` 了。

```
[MapTask.run() > runOldMapper() > PipeMapRunner.run() > PipeMapper.startOutputThreads()]
```

```

PipeMapper.startOutputThreads(OutputCollector output, Reporter reporter)
> InputWriter inWriter_ = createInputWriter()
>> super.createInputWriter(mapInputWriterClass_)
    == PipeMapRed.createInputWriter(mapInputWriterClass_)//外挂进程的标准输入通道
>>> InputWriter inputWriter = ReflectionUtils.newInstance(inputWriterClass, job_)
                                     //创建 TextInputWriter 对象
>>> inputWriter.initialize(this) == TextInputWriter.initialize(this)
>>>> super.initialize(pipeMapRed) == InputWriter.initialize(pipeMapRed)
>>>> clientOut = pipeMapRed.getClientOutput() //获取外挂进程的标准输出通道
>>>>> return clientOut_ //这样,PipeMapper 的 inWriter_,即 TextInputWriter 的 clientOut
                                     //就与外挂进程的 clientOut_对接上了
>>>>> inputSeparator = pipeMapRed.getInputSeparator()
>>>> return inputWriter
> outReader_ = createOutputReader()
>> return super.createOutputReader(mapOutputReaderClass_)
    == PipeMapRed.createOutputReader(mapOutputReaderClass_)
>>> OutputReader outputReader = ReflectionUtils.newInstance(outputReaderClass, job_)
>>> outputReader.initialize(this)
>>> return outputReader
> outThread_ = new MROutputThread(outReader_, output, reporter)
                                     //创建线程与外挂进程的标准输出通道对接
>> setDaemon(true)
>> this.outReader = outReader
>> this.outCollector = outCollector
>> this.reporter = reporter
> outThread_.start()
> errThread_ = new MRErrorThread() //创建线程与挂挂进程的标准出错信息通道对接
> errThread_.setReporter(reporter)
> errThread_.start() //启动

```

总之, PipeMapper 在宿主操作系统上创建一个子进程(或者一个子进程流水线), 然后将一个作为输出流的 InputWriter 对象连接到子进程的 stdin, 并创建两个线程 MROutputThread 和 MRErrorThread, 使它们的输入流分别与子进程的 stdout 和 stderr 对接。这样, PipeMapper 就有了一个挂在体外的数据处理进程, 其作用相当于一个 mapper, 也相当于数据流中的一个节点。而且, 挂在外面的也并不局限于单个进程, 而可以是一段数据流或工作流。进一步, 如果想让数据流在这里分叉, 读者也可以参照 PipeMapper 另写一个, 比方说 SplitMapper, 那样就可以让 Hadoop 支持真正的 DAG, 实现二维的数据流了。

如前所述, PipeMapRunner 是在 MapTask 的线程中运行, 所执行的主循环就是普通 MapRunner 的主循环。每当从其 InputFormat 的 RecordReader 读入一个 KV 对, 就调用 mapper.map(), 但是这时候的 mapper 就是 PipeMapper, 所以被调用的是 PipeMapper.map()。

```
[MapTask.run() > runOldMapper() > PipeMapRunner.run() > MapRunner.run()
> PipeMapper.map()]
```

```
PipeMapper.map(Object key, Object value, OutputCollector output, Reporter reporter)
> numRecRead_ ++
> if (numExceptions_ == 0) { //只要没有出错
>+ if (!this.ignoreKey) inWriter_.writeKey(key)
//将 K 写入 inWriter_ ,这就是外挂 mapper 进程基于其 stdout 通道的输入流
>+ inWriter_.writeValue(value)
//将 V 也写入 inWriter_ ,转交给外挂 mapper 进程
>+ if (skipping) {
>++ clientOut_.flush()
>+ }
> } else {
>+ numRecSkipped_ ++
> }
```

这跟普通 mapper 的 map() 就不一样了, PipeMapper 自己不做 map() 计算, 而只是将 KV 对转交给外挂的 mapper 进程, 在我们这个例子中这就是 grep 进程。换言之, PipeMapRunner 本身的 PipeMapper 成了虚拟的 Mapper, 那个外挂的进程才是实体的 Mapper。

外挂 mapper 进程对输入数据进行处理之后把输出写入它的 stdout 通道, 而 MapTask 创建的子线程 MROutputThread 的输入流就来自这个通道:

```
MROutputThread.run()
> while (outReader.readKeyValue()) { //从输入流读入来自外挂 mapper 进程输出的 KV 对
>+ Object key = outReader.getCurrentKey() //从中读取 K
>+ Object value = outReader.getCurrentValue() //从中读取 V
>+ outCollector.collect(key, value) //将 KV 对写入 Collector, 例如 MapOutputBuffer
//就像普通 mapper 一样
>+ numRecWritten_ ++
>+ long now = System.currentTimeMillis()
>+ if (now - lastStdoutReport > reporterOutDelay_) {
>++ if (!processProvidedStatus_) {
>+++ reporter.setStatus(hline)
>+++ } else {
>+++ reporter.progress()
>+++ }
>+ }
> }
```

这个线程是由 MapTask 通过 PipeMapper 创建的, 是 MapTask 的子线程, 与 MapTask 同在一个 Java 虚拟机上, 共享同一地址空间。所以这个线程所使用的 collector 就是

MapTask 的 collector, 就是同一个 MapOutputBuffer。这样, 在 Reducer 这一边看来一切如常, 根本不知道 Mapper 这一边是在“体外循环”。

而外挂的那个 mapper 进程, 例如 grep, 则当然不在 Java 虚拟机上, 而是与 Java 虚拟机一样都是宿主操作系统上的一个独立进程, 只不过它是 JVM 进程的子进程。

还要指出, 像 grep 这样的 Unix/Linux 程序根本就不知道什么 KV 对, 它只知道输入行和单词, 但是这也不要紧, 如前所述 K 和 V 这二者之一的类型中可以是 NullWritable, 那就相当于空, 就跟不存在一样。

至于外挂 mapper 进程的出错信息, 则都是通过其 stderr 通道输出的, 这些信息来到 MapTask 的另一个子线程 MRErrorThread 的输入流, 这个线程可以将读入的出错信息显示在终端上, 也可以写入日志, 这就比较简单了。

```
MRErrorThread.run()
> lineReader = new LineReader((InputStream)clientErr_, job_)
> while (lineReader.readLine(line) > 0) {
>+ ...
>+ System.err.println(lineStr)
> }
```

MapTask 是这样, ReduceTask 也一样可以外挂。不同的是, Reducer 的数量是在具体 App 程序中设定的, Mapper 的数量则由系统根据输入数据集合的大小和 Split 的最大与最小尺寸计算确定。App 当然无法决定输入数据集的大小, 但却可以通过设置 Split 的最大与最小尺寸来影响 Mapper 的数量。不过需要指出, 用来替代 Mapper 的外挂进程链, 例如“grep | sed”等, 只能存在于同一机器节点上, 因为这些进程之间是靠宿主操作系统的 pipe 机制连起来的, 而 pipe 机制是单机上的进程间通信机制。但是, 另一方面, 每个 Mapper 上都有这么一个外挂进程链, 这是许多外挂进程链的并行计算。

重要的是, 现在可以把本来只能在单机上执行的那些经过千锤百炼的 Unix/Linux 工具程序推广到集群中大规模并行了。

这样, 就把 Hadoop 这个平台与 Unix/Linux 那些 Utility 工具程序乃至程序设计风格都结合起来打成了一片, 真正成了单机操作系统在集群中的延伸。

9.3 Chain 框架

Hadoop 还有一种称为“链(Chain)”的机制, 可以把多个 mapper 串接在一起, 这些 mapper 可以在 Mapper 节点上, 也可以在 Reducer 节点上。显然, 这个机制与上述的 Stream 机制相似, 可是也有很大的不同。在 Chain 中, 被串在一起的那些 mapper 都只能是 Hadoop 的 Mapper 类对象(但可以不同), 而且是运行于同一个 Java 虚拟机上(或者说同一个 JVM 进程中), 而不是像 Stream 那样的外挂和“体外循环”。

对于 Chain 的使用方法, Hadoop 源码中新、老两个 API 分支上的 ChainMapper.java 和 ChainReducer.java 中都有注释。综合新 API 分支 mapreduce 下的这两个文件中的注释, 其使用方法是这样的:


```

...
Job = new Job(conf);
...
Configuration mapAConf = new Configuration(false); //用于第一个 mapper 即 AMap 的 conf
...
ChainMapper.addMapper(job, AMap.class, LongWritable.class, Text.class,
    Text.class, Text.class, true, mapAConf);
//把 AMap 连同其 mapAConf 加到 ChainMapper 的链中
Configuration mapBConf = new Configuration(false); //用于第二个 mapper 即 BMap 的 conf
...
ChainMapper.addMapper(job, BMap.class, Text.class, Text.class,
    LongWritable.class, Text.class, false, mapBConf);
//把 BMap 连同其 mapBConf 加到 ChainMapper 的链中
Configuration reduceConf = new Configuration(false) //用于 Reducer 的 conf
...
ChainReducer.setReducer(job, XReduce.class, LongWritable.class, Text.class,
    Text.class, Text.class, true, reduceConf)
//把 XReduce 设置成 Reducer
ChainReducer.addMapper(job, CMap.class, Text.class, Text.class,
    LongWritable.class, Text.class, false, null)
//把第三个 mapper 即 CMap 加到 ChainReducer 的链中
ChainReducer.addMapper(job, DMap.class, LongWritable.class, Text.class,
    LongWritable.class, LongWritable.class, true, null)
//把第四个 mapper 即 DMap 加到 ChainReducer 的链中
...
job.waitForCompletion(true)
...

```

Hadoop 为此提供了两个类, ChainMapper 和 ChainReducer, 它们可以被用来构筑 Mapper 和 Reducer 两个链。这两个类的对象本身并不提供 map 或 reduce 的功能, 只是在两个链中起着“排头兵”的作用。这里在 ChainMapper 这个链中添上了 AMap 和 BMap 两个 Mapper, 各有自己的配置块 mapAConf 和 mapBConf。而 ChainReducer 这个链则有所不同, 先要设定一个 reducer, 就是这里的 XReduce, 然后在后面加上 CMap 和 DMap 这两个 mapper。

当然, AMap、BMap、CMap、DMap, 还有 XReduce, 都是由用户定义提供的 Mapper 和 Reducer 类。

这就是说, 逻辑概念上的综合的 Mapper 是由 AMap 和 BMap 两个对象构成的链, 数据在 Mapper 阶段要先后经过这两个工位的处理, 从而形成了一段数据流; 而逻辑概念上的 Reducer, 则是在 reducer 即 XReduce 对象后面又加上两个 mapper, 即 CMap 和 DMap, 用来处理 XReduce 的输出, 这三个工位也形成了一段数据流。但是, 在这两段数据流之间则因为 sort 的存在而变成了工作流, 就像中间有个“拦河坝”。

明白了这个思路和原理,我们先看 ChainMapper。下面是 ChainMapper 类的定义摘要:

```
class ChainMapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
    extends Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {}

] Chain chain

] addMapper(Job job, Class<?extends Mapper> klass,
    Class<?> inputKeyClass, Class<?> inputValueClass,
    Class<?> outputKeyClass, Class<?> outputValueClass,
    Configuration mapperConf)

> job.setMapperClass(ChainMapper.class) //就作业而言,所用的 Mapper 是 ChainMapper
> job.setMapOutputKeyClass(outputKeyClass) //作为整个 Mapper 的 K 输出类型
> job.setMapOutputValueClass(outputValueClass) //作为整个 Mapper 的 V 输出类型
> Chain.addMapper(true, job, klass, inputKeyClass, inputValueClass,
    outputKeyClass, outputValueClass, mapperConf)
== Chain.addMapper(boolean isMap, Job job, Class<?extends Mapper> klass,
    Class<?> inputKeyClass, Class<?> inputValueClass,
    Class<?> outputKeyClass, Class<?> outputValueClass,
    Configuration mapperConf)

>> String prefix = getPrefix(isMap)
>>> return (isMap)?CHAIN_MAPPER : CHAIN_REDUCER
    //“mapreduce.chain.mapper”

>>> Configuration jobConf = job.getConfiguration()
>> checkReducerAlreadySet(isMap, jobConf, prefix, true) //对 Map 阶段没有影响
>> index = getIndex(jobConf, prefix) //是 Mapper 链中的第几个 Mapper
>>> return conf.getInt(prefix + CHAIN_MAPPER_SIZE, 0)
>> jobConf.setClass(prefix + CHAIN_MAPPER_CLASS + index, klass, Mapper.class)
    //把 Mapper 链中的第几个 Mapper 设置成给定的类,例如 AMap
>> validateKeyValueTypes(isMap, jobConf, inputKeyClass, inputValueClass,
    outputKeyClass, outputValueClass, index, prefix)
    //检查这个 Mapper 的输入类型与前一个 Mapper 的输出是否匹配
>> setMapperConf(isMap, jobConf, inputKeyClass, inputValueClass,
    outputKeyClass, outputValueClass, mapperConf, index, prefix)
    //将给定的 mapperConf 设置成这个 Mapper 的 conf

] setup(Context context)
    > chain = new Chain(true)
    > chain.setup(context.getConfiguration())

] run(Context context) //投运时才实际构成 Mapper 链
```

ChainMapper 类是对 Mapper 类的扩充,它定义了自己的 setup()和 run(),增加了一个方法 addMapper()。它还增加了一个 Chain 类的对象 chain,不过这个对象要到投运的时候才会在 setup()中创建。

Chain 类属于 org.apache.hadoop.mapreduce.lib.chain 这个 package, 其完整的定义在文件 mapreduce/lib/chain/Chain.java 中, 下面将只介绍这里要用到的几个函数。

完成了对于 Mapper 链(以及 Reducer 链)的描述之后, 就是常规的作业提交, 只不过作业的 Mapper 是 ChainMapper。而由用户提供的那些实际的 Mapper 如 AMap、BMap 之类, 则相当于 Mapper 依赖和用到的其他的类, 对系统而言属于用户提供的资源。

到了 Mapper 要在具体的 NM 节点上投运的时候, MapTask 在其 runNewMapper() 中通过 taskContext.getMapperClass() 获取具体的 Mapper 类, 此时为 ChainMapper 类, 然后通过 ReflectionUtils.newInstance() 创建该类对象, 最后调用 mapper.run(), 这就调用到了 ChainMapper.run():

```
[MapTask.run() > runNewMapper() > mapper.run()]
```

```
ChainMapper.run()
> setup(context)
>> chain = new Chain(true) //创建 Chain 对象
>>> chain.setup(context.getConfiguration())
//根据 JobConf 中的内容依次创建链中的各个 Mapper 对象:
>>>> String prefix = getPrefix(isMap)
>>>> index = jobConf.getInt(prefix + CHAIN_MAPPER_SIZE, 0)
>>>> for (int i = 0; i < index; i++) {
>>>>+ Class<?extends Mapper> klass =
>>>>+ jobConf.getClass(prefix + CHAIN_MAPPER_CLASS + i, null, Mapper.class)
>>>>+ Configuration mConf = getChainElementConf(
>>>>+ jobConf, prefix + CHAIN_MAPPER_CONFIG + i)
>>>>+ confList.add(mConf)
>>>>+ Mapper mapper = ReflectionUtils.newInstance(klass, mConf)
>>>>+ //创建实际的 Mapper 对象, 例如 AMap, BMap
>>>>+ mappers.add(mapper) //将创建的 Mapper 对象加入 mappers 序列
>>>> }
>>>> Class<?extends Reducer> klass =
>>>>+ jobConf.getClass(prefix + CHAIN_REDUCER_CLASS, null, Reducer.class)
>>>> if (klass != null) {
>>>>+ rConf = getChainElementConf(jobConf, prefix + CHAIN_REDUCER_CONFIG)
>>>>+ reducer = ReflectionUtils.newInstance(klass, rConf) //如果设定了 Reducer 则创建对象
>>>> }
> numMappers = chain.getAllMappers().size() //链中共有几个 Mapper
> if (numMappers == 0) return
> if (numMappers == 1) {
>+ chain.runMapper(context, 0) //只有一个实际的 mapper, 那就运行这个 Mapper
> } else { //是多个 mapper 的链
>+ ChainBlockingQueue<Chain.KeyValuePair<?, ?>> outputqueue =
```

```

chain.createBlockingQueue()
//创建第一个 Mapper 的输出队列
>+ chain.addMapper(context, outputqueue, 0) //将第一个 mapper 加入 mapper 链
>+ for (int i=1; i < numMappers-1; i++) { //除第一个和最后一个 mapper 之外的中间节点:
>+ inputqueue = outputqueue
//前一个 mapper 的输出队列就是后一个 mapper 的输入队列
>+ outputqueue = chain.createBlockingQueue() //再创建后一个 Mapper 的输出队列
>+ chain.addMapper(inputqueue, outputqueue, context, i)
//将后一个 mapper 加入 mapper 链
>+ }
>+ chain.addMapper(outputqueue, context, numMappers-1)
//add last mapper,将最后一个 Mapper 加入 Mapper 链,这个 mapper 无须输出队列
> }
> chain.startAllThreads() //start all threads,启动链中的所有线程
> chain.joinAllThreads() //wait for all threads,等待链中的所有线程结束

```

我在这里已经添加了足够详尽的注释,读者在理解时应该不会有困难。但是请注意,这里有三处对 `Chain.addMapper()` 的调用,加上前面提交作业时的那次,就有四处了。这四处调用,函数的名称都叫 `addMapper()`,但是调用参数的序列却各不相同,实际上是四种不同的操作。第一次,就是在提交作业时的那次,有 8 个参数,那是对 Mapper 链的描述和设置;其余三处都是在投运的时候用来构成实际的 Mapper 链,但是针对链中的第一个 mapper、作为中间节点的 mapper 和最后一个 mapper 的操作又各不相同。其中用于第一个 mapper 的 `addMapper()` 只有 3 个参数,因为第一个 mapper 无所谓输入队列,它的输入就是通过整个 Mapper 的 `InputFormat` 的 `RecordReader` 读入的。作为中间节点的那些 mapper,则需要承前启后,有输入队列也有输出队列,所以需要 4 个参数。而最后一个 mapper,则无所谓输出队列,它的输出就是整个 Mapper 的 `MapOutputCollector`,所以也只要 3 个参数,但是这 3 个参数不同于第一个 mapper 的那 3 个参数。函数名一样都是 `addMapper()`,只是靠参数序列的不同而区分,这就是“多态(Polymorphism)”。多态当然很有好处,但是从阅读、分析程序的角度,我倒宁可这些函数的名称分别是比方说 `addMapper()`、`linkFirstMapper()`、`linkLastMapper()`,等等。

下面我们看一下这三个不同的 `addMapper()`。首先是针对 Chain 中的第一个 mapper:

```
[MapTask.run() > runNewMapper() > ChainMapper.run() > Chain.addMapper()]
```

```

Chain.addMapper(TaskInputOutputContext inputContext,
ChainBlockingQueue<KeyValuePair<?,?>> output, int index)
//将第一个 mapper 加入 Mapper 链
> Configuration conf = getConf(index)
> Class<?> keyOutClass = conf.getClass(MAPPER_OUTPUT_KEY_CLASS, Object.class)
> Class<?> valueOutClass = conf.getClass(MAPPER_OUTPUT_VALUE_CLASS, Object.class)
> RecordReader rr = new ChainRecordReader(inputContext)

```

```

//创建第一个 mapper 的 RecordReader,取决于 inputContext 中的设置
> RecordWriter rw = new ChainRecordWriter(keyOutClass, valueOutClass, output, conf)
//创建第一个 mapper 的 RecordWriter,通往其输出队列 output
> Mapper.Context mapperContext = createMapContext(rr, rw,
                                                    (MapContext) inputContext, getConf(index))
> MapRunner runner = new MapRunner(mappers.get(index), mapperContext, rr, rw)
//创建这第一个 mapper 的 MapRunner 线程
> threads.add(runner) //将此线程加入这个 Chain 的线程集合 threads

```

注意,这里创建了一个 MapRunner 线程。对比前述的 Stream,那里的 mapper 是个独立的进程,而这里在 Chain 机制中则是与 MapTask 同在一个 JVM 上的线程。

再看针对中间节点的 addMapper():

```
[MapTask.run() > runNewMapper() > ChainMapper.run() > Chain.addMapper()]
```

```

Chain.addMapper(ChainBlockingQueue<KeyValuePair<?,?>> input,
                ChainBlockingQueue<KeyValuePair<?,?>> output,
                TaskInputOutputContext context, int index)
//将作为中间节点的 Mapper 加入 mapper 链
> Configuration conf = getConf(index)
> Class<?> keyClass = conf.getClass(MAPPER_INPUT_KEY_CLASS, Object.class)
> Class<?> valueClass = conf.getClass(MAPPER_INPUT_VALUE_CLASS, Object.class)
> Class<?> keyOutClass = conf.getClass(MAPPER_OUTPUT_KEY_CLASS, Object.class)
> Class<?> valueOutClass = conf.getClass(MAPPER_OUTPUT_VALUE_CLASS, Object.class)
> RecordReader rr = new ChainRecordReader(keyClass, valueClass, input, conf)
> RecordWriter rw = new ChainRecordWriter(keyOutClass, valueOutClass, output, conf)
> MapRunner runner = new MapRunner(mappers.get(index), createMapContext(rr,
                                                                    rw, context, getConf(index)), rr, rw)
//创建这个中间 mapper 的 MapRunner 线程
> threads.add(runner) //将此线程加入这个 Chain 的线程集合 threads

```

这里也要创建 MapRunner 线程。这个线程的输入来自其输入队列,那就是前一个节点的输出队列。

最后,是针对最后一个 mapper 节点的 addMapper():

```
[MapTask.run() > runNewMapper() > ChainMapper.run() > Chain.addMapper()]
```

```

Chain.addMapper(ChainBlockingQueue<KeyValuePair<?,?>> input,
                TaskInputOutputContext outputContext, int index)
//将最后一个 mapper 加入 Mapper 链
> Configuration conf = getConf(index)
> Class<?> keyClass = conf.getClass(MAPPER_INPUT_KEY_CLASS, Object.class)
> Class<?> valueClass = conf.getClass(MAPPER_INPUT_VALUE_CLASS, Object.class)

```

```

> RecordReader rr = new ChainRecordReader(keyClass, valueClass, input, conf)
> RecordWriter rw = new ChainRecordWriter(outputContext)
//最后一个 mapper 的输出去往 MapOutputCollector
> MapRunner runner = new MapRunner(mappers.get(index), createMapContext(rr,
    rw, outputContext, getConf(index)), rr, rw)
//创建这最后一个 mapper 的 MapRunner 线程
> threads.add(runner) //将此线程加入这个 Chain 的线程集合 threads

```

搭建好了 Mapper 链,就通过 Chain.startAllThreads()启动这段链上的所有 MapRunner 线程,让它们各自进入其 run()函数:

```

class MapRunner<KEYIN, VALUEIN, KEYOUT, VALUEOUT> extends Thread {}
] Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> mapper
] Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>.Context chainContext
] RecordReader<KEYIN, VALUEIN> rr
] RecordWriter<KEYOUT, VALUEOUT> rw
] MapRunner(Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> mapper,
    Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>.Context mapperContext,
    RecordReader<KEYIN, VALUEIN> rr, RecordWriter<KEYOUT, VALUEOUT> rw)
] run()
> if (getThrowable() != null) return
> mapper.run(chainContext) == Mapper.run() //这个 Mapper 就是 AMap、BMap 之类
>> setup(context)
>> while (context.nextKeyValue()) {
>>+ map(context.getCurrentKey(), context.getCurrentValue(), context)
>> }
>> cleanup(context)
> rr.close()
> rw.close(chainContext)

```

每个 MapRunner 的 run() 函数都调用其 mapper 的 run(), 那就是用户提供的诸如 AMap、BMap 之类 Mapper 的 run()。于是,这段 Chain 就作为一段数据流在运行了。这是因为,每个节点从其输入端读入一个 KV 对,经过处理之后就写了出去,成为下一个节点的输入,数据传输的粒度是一个 KV 对。

数据的传输还有“按值(by value)”和“按引用(by reference)”之分,这是可以设置的。前者在传输时要进行 KV 对的复制,是传变量(缓冲区)的值而不是变量的地址,这是“正宗”函数式程序设计的做法,正确性好但效率稍低;后者则是传缓冲区的地址,效率高但容易出问题。当然,之所以可以按传地址的方式工作,是因为这些 mapper 都在同一个 JVM 进程中。

搞懂了 Mapper 链,Reducer 链就不难理解了,我把这留给读者自己。

9.4 Client 与 ApplicationMaster

Hadoop 为使用者提供了一种类似于 Shell 的命令行用户界面,由一个称为 Client 的类加

以实现。注意,这跟 RPC 层上的 Client 毫无关系,是两码事。这个类的对象可以被用来向 YARN 框架提交作业,所以也应该算是 YARN 框架的一部分。与前述 MapReduce 作业的提交不同,通过 Client 提交的通常并非 MapReduce 作业,而可以是按用户自行设计的模型开发的作业和任务,例如可以是像 Unix/Linux 的 Shell 作业流水线那样的一维数据流(然而又不同于 Shell,因为所形成的流水线可以重复很多份,分布在集群中从事并行计算)。所以这个框架比 MapReduce 更为一般化,其设计思路从某些角度看更为通用,不像 MapReduce 那样专门。

这个 Client 类的定义摘要是这样的:

```
class Client {} //src/main/yarn/applications/distributedshell
] Configuration conf
] YarnClient yarnClient
] int containerMemory = 10
] int containerVirtualCores = 1
] int numContainers = 1
] Client(String appMasterMainClass, Configuration conf)
] main()
] init(String[] args)
] run()
] monitorApplication(ApplicationId appId)
```

Client 是具有 main() 函数,可以独立作为 JVM 进程运行的。我们从它的 main() 函数开始。

```
Client.main()
> this.conf = conf
> client = new Client()
>> this.appMasterMainClass =
    "org.apache.hadoop.yarn.applications.distributedshell.ApplicationMaster"
>> YarnClient yarnClient = YarnClient.createYarnClient()
>> yarnClient.init(conf)
>> opts = new Options()
>> opts.addOption("appname", true, "Application Name. Default value - DistributedShell")
>> ... //许多选项
>> opts.addOption("master_memory", true,
    "Amount of memory in MB to be requested to run the application master")
>> opts.addOption("master_vcores", true,
    "Amount of virtual cores to be requested to run the application master")
>> opts.addOption("shell_command", true, "Shell command to be executed by " +
    "the Application Master. Can only specify either -- shell_command " +
    "or -- shell_script")
>> opts.addOption("shell_script", true, "Location of the shell script to be " +
```

```

        "executed. Can only specify either -- shell_command or -- shell_script")
>> opts.addOption("shell_args", true, "Command line args for the shell script." +
        "Multiple args can be separated by empty space.")
>> ... //还有许多选项,限于篇幅不予列举
> client.init(args)
> client.run() == Client.run()

```

继续往下看它的 run() 函数:

```

Client.run()
> yarnClient.start()
> YarnClientApplication app = yarnClient.createApplication() //向 RM 请求建立一个 App
>> ApplicationSubmissionContext context =
        Records.newRecord(ApplicationSubmissionContext.class)
        //创建一个 ApplicationSubmissionContext,即 ASC
>> GetNewApplicationResponse newApp = getNewApplication()
        //向 RM 发送请求并等待回应
>> ApplicationId appId = newApp.getApplicationId() //RM 的回应中包含 ApplicationId
>> context.setApplicationId(appId) //将 ApplicationId 设置在 ASC 中
>> return new YarnClientApplication(newApp, context)
        //返回据此而创建的 YarnClientApplication 对象
> maxMem = appResponse.getMaximumResourceCapability().getMemory()
        //RM 承诺拨给的内存
> maxVCores = appResponse.getMaximumResourceCapability().getVirtualCores() //和 VCore
> ApplicationSubmissionContext appContext = app.getApplicationSubmissionContext()
> appId = appContext.getApplicationId()
> appContext.setKeepContainersAcrossApplicationAttempts(keepContainers)
> appContext.setApplicationName(appName)
> ...
        //构筑准备让 RM 指定和安排在某个 NM 节点上启动的 Shell 命令行
> vargs = new Vector<CharSequence>(30)
> vargs.add(Environment.JAVA_HOME.$$(()) + "/bin/java") //首先,要启动的是 Java
> vargs.add("-Xmx" + amMemory + "m")
> vargs.add(appMasterMainClass) //要求 Java 虚拟机运行的是 ApplicationMaster
> ...
> vargs.add("1>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/AppMaster.stdout")
> vargs.add("2>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/AppMaster.stderr")
> command = new StringBuilder()
> for (CharSequence str : vargs) command.append(str).append(" ") //完成命令行的构筑
> ContainerLaunchContext amContainer =
        ContainerLaunchContext.newInstance(localResources, env, commands, null, null, null)

```

```

//创建 CLC
> appContext.setAMContainerSpec(amContainer) // amContainer 是 CLC,并非 Container
> ...
> yarnClient.submitApplication(appContext)
    == YarnClientImpl.submitApplication(appContext) //向 RM 提交作业请求
>> applicationId = appContext.getApplicationId()
>> SubmitApplicationRequest request = Records.newRecord(SubmitApplicationRequest.class)
>> request.setApplicationSubmissionContext(appContext)
>> rmClient.submitApplication(request)
> stat = monitorApplication(appId)
> return stat

```

注意这个 run()函数里面没有通常可以见到的那种 while 循环,所以这些代码只被执行一次,目的就是提交一个 App,并等待其完成。但是在提交了 App 以后的 monitorApplication() 中有个 while 循环,过一会儿就报告一下进展,直至 App 运行结束,此时程序从 run()返回到 main(),接着就退出了。下一次如果又要提交一个作业,就又在宿主操作系统的提示符下启动 Java 虚拟机执行 Client。

通过 Client 提交作业的直接效果,就是 RM 在某个 NM 节点上启动一个 Java 虚拟机,运行客户要求的 Java 类。以前在 MapReduce 框架上那是 MRAppMaster,而现在则变成了 ApplicationMaster。这对于 RM 而言是透明的,RM 并不关心你要运行的是什么,它只是找一个 NM 节点,让它运行你提交的 Shell 命令行。之所以这个命令行中指定运行 bin/java,是因为所提交的 ApplicationMaster(以前是 MRAppMaster)是个 Java 类。YARN 框架就是这样设计的,你得按它的规矩办才能利用 YARN 所提供的许多功能。至于这个 Java 类运行起来之后是否变成“项目组长”,是否把你的 App 分解成任务并向 RM 协商分配资源,以及如何把这些任务投运,那就看这个 Java 类的设计和实现了。ApplicationMaster 的性质和作用与 MRAppMaster 相近,但是互相没有什么关系。

总之,在 Client 的要求下,在集群中的某个 NM 节点上会启动一个 Java 虚拟机 JVM,运行 ApplicationMaster。我们先看一下它的大致构造,然后再顺着程序流程细看。

```

class ApplicationMaster {} //yarn/applications/distributedshell/ApplicationMaster.java
] AMRMClientAsync amRMClient // 实际上是 AMRMClientAsyncImpl,与 RM 交互
] UserGroupInformation appSubmitterUgi
] NMClientAsync nmClientAsync //实际上是 NMClientAsyncImpl,与 NM 交互
] NMCallbackHandler containerListener
] String ExecShellStringPath = Client.SCRIPT_PATH + ".sh"
] String shellCommandPath = "shellCommands"
] String linux_bash_command = "bash"
] List<Thread> launchThreads = new ArrayList<Thread>()
] main(String[] args)
] run()
] class RMCallbackHandler implements AMRMClientAsync.CallbackHandler {}

```

```

//负责处理 RM 节点发回的响应
]] onContainersCompleted(List<ContainerStatus> completedContainers)
]] onContainersAllocated(List<Container> allocatedContainers)
]] class NMCallbackHandler implements NMClientAsync.CallbackHandler {}
//负责来自 NM 节点的事件报告
]] onContainerStatusReceived(ContainerId containerId, ContainerStatus containerStatus)
]] onContainerStarted(ContainerId containerId, Map<String, ByteBuffer> allServiceResponse)
]] class LaunchContainerRunnable implements Runnable {}
//负责容器投运
]] run()

```

应 Client 要求在 NM 节点上启动运行 ApplicationMaster 时,程序入口是它的 main() 函数:

```

ApplicationMaster.main(String[] args)
> appMaster = new ApplicationMaster()
>> conf = new YarnConfiguration()
> appMaster.init(args) == ApplicationMaster.init(args)
>> Map<String, String> envs = System.getenv()
>> if (!envs.containsKey(Environment.CONTAINER_ID.name())) {
>>+ if (cliParser.hasOption("app_attempt_id")) {
>>++ String appIdStr = cliParser.getOptionValue("app_attempt_id", "")
>>++ appAttemptID = ConverterUtils.toApplicationAttemptId(appIdStr)
>>+ } else {
>>++ throw new IllegalArgumentException("Application Attempt Id not set in the environment")
>>+ }
>> } else {
>>+ containerId =
    ConverterUtils.toContainerId(envs.get(Environment.CONTAINER_ID.name()))
>>+ appAttemptID = containerId.getApplicationAttemptId()
>> }
>> if (fileExist(shellCommandPath)) shellCommand = readContent(shellCommandPath)
>> if (fileExist(shellArgsPath)) shellArgs = readContent(shellArgsPath)
>> ...
>> containerMemory = Integer.parseInt(cliParser.getOptionValue("container_memory", "10"))
>> containerVirtualCores = Integer.parseInt(cliParser.getOptionValue("container_vcores", "1"))
>> numTotalContainers = Integer.parseInt(cliParser.getOptionValue("num_containers", "1"))
> appMaster.run() == ApplicationMaster.run()
> appMaster.finish()

```

初始化以后,程序就进入 ApplicationMaster 的 run() 函数:

```
[ApplicationMaster.main() > ApplicationMaster.run()]
```

```

ApplicationMaster.run()
> dob = new DataOutputStream()
> AMRMClientAsync.CallbackHandler allocListener = new RMCallbackHandler()
> amRMClient = AMRMClientAsync.createAMRMClientAsync(1000, allocListener)
//建立对 RM 进行 RPC 的客户端和 proxy
>> return new AMRMClientAsyncImpl<T>(intervalMs, callbackHandler)
//创建 AMRMClientAsyncImpl 对象,这是一种 Service
>>> super(client, intervalMs, callbackHandler) == AMRMClientAsync(...)
>>>> super(AMRMClientAsync.class.getName()) == AbstractService(...)
>>>> this.client = client
>>>> this.heartbeatIntervalMs.set(intervalMs)
>>>> this.handler = callbackHandler;
>>> heartbeatThread = new HeartbeatThread() //创建 HeartbeatThread 线程
>>> handlerThread = new CallbackHandlerThread() //创建 CallbackHandlerThread 线程
>>> responseQueue = new LinkedBlockingQueue<AllocateResponse>()
>>> keepRunning = true
> amRMClient.init(conf) == AbstractService.init(conf)
>> AMRMClientAsyncImpl.serviceInit(config)
>>> super.serviceInit(conf)
>>> client.init(conf) == AMRMClientAsyncImpl.init(conf)
> amRMClient.start() == AbstractService.start()
>> AMRMClientAsyncImpl.serviceStart()
>>> handlerThread.setDaemon(true)
>>> handlerThread.start() //启动 CallbackHandlerThread 线程
>>> client.start()
>>> super.serviceStart()
> containerListener = createNMCallbackHandler()
>> return new NMCallbackHandler(this)
> nmClientAsync = new NMClientAsyncImpl(containerListener)
//创建 NMClientAsyncImpl 对象,也是一种 Service
> nmClientAsync.init(conf)
> nmClientAsync.start()
>> NMClientAsyncImpl.serviceStart()
>>> client.start()
>>>> ThreadFactory tf = new ThreadFactoryBuilder().setNameFormat(
this.getClass().getName() + " # %d").setDaemon(true).build()
>>>> initSize = Math.min(INITIAL_THREAD_POOL_SIZE, maxThreadPoolSize)
>>>> threadPool = new ThreadPoolExecutor(initSize, Integer.MAX_VALUE, 1,
TimeUnit.HOURS, new LinkedBlockingQueue<Runnable>(), tf)

```

```

>>> eventDispatcherThread = new Thread()
>>> eventDispatcherThread.setDaemon(false)
>>> eventDispatcherThread.start()
>>> super.serviceStart()
> response = amRMClient.registerApplicationMaster(
    appMasterHostname, appMasterRpcPort, appMasterTrackingUrl)
>> response = client.registerApplicationMaster(appHostName, appHostPort, appTrackingUrl)
    //向 RM 节点上的 ApplicationMasterService 登记
>> heartbeatThread.start()    //开始发送心跳报告
> maxMem = response.getMaximumResourceCapability().getMemory()
    //RM 节点承诺的最大内存空间
> maxVCores = response.getMaximumResourceCapability().getVirtualCores()
    //RM 节点承诺的 VCore 数量
> List<Container> previousAMRunningContainers =
    response.getContainersFromPreviousAttempts()
    //上一次尝试留下来的容器数量
> numAllocatedContainers.addAndGet(previousAMRunningContainers.size())
> numTotalContainersToRequest =
    numTotalContainers - previousAMRunningContainers.size()
    //尚需申请的容器数量,容器总数 numTotalContainers 来自用户命令行
> for (int i = 0; i < numTotalContainersToRequest; ++i) {    //对每一个尚需申请的容器:
>+ ContainerRequest containerAsk = setupContainerAskForRM()    //准备一份申请
>+ request = new ContainerRequest(capability, null, null, pri)
>+>> ContainerRequest(Resource capability, String[] nodes, String[] racks, Priority priority)
>+> return request
>+ amRMClient.addContainerRequest(containerAsk)    //先聚集在一起
>+> client.addContainerRequest(req)
> }
> numRequestedContainers.set(numTotalContainers)
> publishApplicationAttemptEvent(timelineClient, appAttemptID.toString(),
    DSEvent.DS_APP_ATTEMPT_END, domainId, appSubmitterUgi)

```

作为具体 App 作业的“项目组长”，AM 即 ApplicationMaster 的作用无非就是向上申请资源，向下管理各个组员的执行情况。向上的这面是一个 AMRMClientAsyncImpl 对象，这里有两个线程，一个是 HeartbeatThread，另一个是 CallbackHandlerThread。前者负责向上报告情况和要求分配资源；后者负责处理上级发回的响应。而向下的这一面，则是一个 NMClientAsyncImpl 对象，那里面有一个线程 eventDispatcherThread，同时还有一个线程池 threadPool。之所以要有一个线程池，是因为 AM 的下面一般都有很多个节点在执行这个 App 中的任务。

AMRMClientAsyncImpl.HeartbeatThread 线程每过一会儿就向 RM 要求 allocate()：


```
AMRMClientAsyncImpl.HeartbeatThread.run()
```

```
> while (true) {
>+ AllocateResponse response = client.allocate(progress)
    == AMRMClientImpl.allocate(float progressIndicator) //请求分配资源
>+ if (response != null)
>++ responseQueue.put(response)
    //如果有回应就将其挂入队列,由 CallbackHandlerThread 加以处理
>+ }
>+ Thread.sleep(heartbeatIntervalMs.get()) //睡一会儿
> }
```

不过说是 allocate(), 倒也不是瞎要, 向 RM 开列的具体要求来自一个集合 ask, 这个集合中的资源要求是通过 addResourceRequestToAsk() 加进去的, 最终还是来自 RM 交下来的作业。但是不管 ask 集合中是否还有资源要求, allocate 请求还是过一会儿就要发一下, 这就像心跳一样。

```
[AMRMClientAsyncImpl.run() > AMRMClientImpl.allocate()]
```

```
AMRMClientImpl.allocate(float progressIndicator)
```

```
> askList = new ArrayList<ResourceRequest>(ask.size())
> for(ResourceRequest r : ask) { //把 ask 集合中的资源要求转移到 askList 中
>+ askList.add(ResourceRequest.newInstance(r.getPriority(),
    r.getResourceName(), r.getCapability(), r.getNumContainers(),
    r.getRelaxLocality(), r.getNodeLabelExpression()))
> }
> releaseList = new ArrayList<ContainerId>(release) //把已可释放的资源名单转到 releaseList
> ask.clear()
> release.clear()
> allocateRequest = AllocateRequest.newInstance(lastResponseId, progressIndicator,
    askList, releaseList, blacklistRequest) //创建一个分配请求
> allocateResponse = rmClient.allocate(allocateRequest) //通过 RPC 调用请求 RM 分配资源
> clusterNodeCount = allocateResponse.getNumClusterNodes()
> lastResponseId = allocateResponse.getResponseId()
> clusterAvailableResources = allocateResponse.getAvailableResources()
> if (!allocateResponse.getNMTokens().isEmpty()) {
>+ populateNMTokens(allocateResponse.getNMTokens())
> }
> if (allocateResponse.getAMRMToken() != null) {
>+ updateAMRMToken(allocateResponse.getAMRMToken())
> }
> if (!pendingRelease.isEmpty()
    && !allocateResponse.getCompletedContainersStatuses().isEmpty()) {
```

```

>+ removePendingReleaseRequests(allocateResponse.getCompletedContainersStatuses())
> }
> if(allocateResponse == null) { //如果得不到响应
>+ release.addAll(releaseList) //把 releaseList 中的可释放资源清单放回 release 集合
>+ for(ResourceRequest oldAsk : askList) {
>++ if(!ask.contains(oldAsk)) {
>+++ ask.add(oldAsk) //把 askList 中的资源要求放回 ask 集合,下次再试
>++ }
>+ }
> }
> return allocateResponse

```

如前所述, RM 节点上有专门为 AM 提供服务的 ApplicationMasterService, 这里所做的 RPC 调用实际上是对 ApplicationMasterService.allocate() 的调用。我们在前面看到, MRAppMaster 也是这样向 RM 申请分配资源的, 现在换成了 ApplicationMaster, 但是 ApplicationMasterService 并不关心对方是谁, 只要按规定的界面提出申请就行。

具体的资源分配取决于调度器, 这里假定 RM 所用的是最简单的 FifoScheduler, 我们在前一章中对此已有叙述, 这里就不再解释了。

ApplicationMaster 跟 MRAppMaster 一样是“中层干部”, 上要向 RM 报告, 向 RM 索要资源, 下则要管理所辖 Task 所在的那些 NodeManager 上的“基层干部”, 所以有一个 eventDispatcherThread 线程, 专门与这些节点打交道:

```

NMClientAsyncImpl.eventDispatcherThread.run()
> Set<String> allNodes = new HashSet<String>()
> while (!stopped.get() && !Thread.currentThread().isInterrupted()) {
>+ event = events.take()
>+ allNodes.add(event.getNodeId().toString())
>+ threadPoolSize = threadPool.getCorePoolSize()
>+ if (threadPoolSize != maxThreadPoolSize) {
>++ nodeNum = allNodes.size() //这不是集群的大小, 而只是与这个 AM 有关的节点的数量
>++ idealThreadPoolSize = Math.min(maxThreadPoolSize, nodeNum)
>++ if (threadPoolSize < idealThreadPoolSize) {
>+++ newThreadPoolSize = Math.min(maxThreadPoolSize,
>                                     idealThreadPoolSize + INITIAL_THREAD_POOL_SIZE)
>+++ threadPool.setCorePoolSize(newThreadPoolSize)
>++ }
>+ } //end if (threadPoolSize != maxThreadPoolSize)
>+ ContainerEventProcessor proc = getContainerEventProcessor(event)
>+ return new ContainerEventProcessor(event)
>+ threadPool.execute(proc) == ContainerEventProcessor.run()
>+ ContainerId containerId = event.getContainerId();

```

```

>+> if (event.getType() == ContainerEventType.QUERY_CONTAINER) {
>+>+ ContainerStatus containerStatus =
            client.getContainerStatus(containerId, event.getNodeId());
>+>+ callbackHandler.onContainerStatusReceived(containerId, containerStatus)
>+> } else {
>+>+ StatefulContainer container = containers.get(containerId)
>+>+ if (container == null) {
>+>+ LOG.info("Container " + containerId + " is already stopped or failed")
>+>+ } else {
>+>+ container.handle(event) == StatefulContainer.handle(event) //驱动其状态机
>+>+ if (isCompletelyDone(container)) {
>+>+ containers.remove(containerId)
>+>+ }
>+> } //end if (container == null) ... else ...
>+> } //end if (QUERY_CONTAINER) ... else ...
> } //end while

```

向上则每隔一段时间就要向 RM 发送心跳报告,这也有个 HeartbeatThread 线程专管。

```

AMRMClientAsyncImpl.HeartbeatThread.run()
> while (true) {
>+ AllocateResponse response = client.allocate(progress)
            == AMRMClientImpl.allocate(float progressIndicator)
>+ if (response != null) {
>+ while (true) {
>+ responseQueue.put(response); //挂入 responseQueue 队列
>+ break;
>+ }
>+ }
>+ Thread.sleep(heartbeatIntervalMs.get())
> } //end while

```

所谓心跳,在这儿就是 allocate()。

不过这个线程自己并不直接处理来自 RM 的响应,而是把响应信息挂在 responseQueue 队列中,让另一个 CallbackHandlerThread 线程加以处理:

```

AMRMClientAsyncImpl.CallbackHandlerThread.run()
> while (true) {
>+ response = responseQueue.take(); //从 responseQueue 队列取出
>+ List<NodeReport> updatedNodes = response.getUpdatedNodes();
>+ if (!updatedNodes.isEmpty()) {
>+ handler.onNodesUpdated(updatedNodes); //有节点的状态发生变化,做出反应
>+ }

```

```

>+ List<ContainerStatus> completed = response.getCompletedContainersStatuses();
//也许有容器已经完成运行
>+ if (!completed.isEmpty()) {
>+ handler.onContainersCompleted(completed); //有容器已经完成运行,做出反应
>+ }
>+ List<Container> allocated = response.getAllocatedContainers(); //也许分配到了容器
>+ if (!allocated.isEmpty()) {
>+ handler.onContainersAllocated(allocated); //分配到了容器,做出反应
>+ }
>+ progress = handler.getProgress();
> } //end while

```

上面程序中的 handler 就是 RMCallbackHandler, 这个 handler 专门处理来自 RM 的事件, 其中最重要的当然是分配到了容器, 此时就为分配到的每一个容器都创建一个 LaunchContainerRunnable, 安排一个线程来专门处理这个容器的投运:

```

class RMCallbackHandler implements AMRMClientAsync.CallbackHandler {}
] onContainersCompleted(List<ContainerStatus> completedContainers)
] onContainersAllocated(List<Container> allocatedContainers)
  > for (Container allocatedContainer : allocatedContainers)
    >> runnableLaunchContainer =
        new LaunchContainerRunnable(allocatedContainer, containerListener)
    >> launchThread = new Thread(runnableLaunchContainer)
    >> launchThreads.add(launchThread)
    >> launchThread.start()
  > }

```

从 RM 那里拿到容器, AM 的职责就是容器的投运。容器的投运要专门为其创建一个 LaunchContainerRunnable 线程, 因为这不是一个“立等可取”的事情, 不能让主线程干等。

```

ApplicationMaster.LaunchContainerRunnable.run()
> Map<String, LocalResource> localResources = new HashMap<String, LocalResource>();
> if (!scriptPath.isEmpty()) {
>+ if (Shell.WINDOWS) {
>+ renamedScriptPath = new Path(scriptPath + ".bat"); //在 Windows 上,脚本就是.bat 文件
>+ } else {
>+ renamedScriptPath = new Path(scriptPath + ".sh"); //在 Linux 上,脚本就是.sh 文件
>+ }
>+ renameScriptFile(renamedScriptPath);
>+ URL yarnUrl = ConverterUtils.getYarnUrlFromURI(
    new URI(renamedScriptPath.toString()));
>+ LocalResource shellRsrc = LocalResource.newInstance(yarnUrl,
    LocalResourceType.FILE, LocalResourceVisibility.APPLICATION,

```

```

        shellScriptPathLen, shellScriptPathTimestamp);
>+ localResources.put(Shell.WINDOWS ?ExecBatScripStringtPath
                        : ExecShellStringPath, shellRsrc);
>+ shellCommand = Shell.WINDOWS ?windows_command : linux_bash_command;
                        //对于 Linux 是“bash”,对于 windows 是“cmd /c”
> } //end if (!scriptPath.isEmpty())
> Vector<CharSequence> vargs = new Vector<CharSequence>(5); //进一步构建命令行
> vargs.add(shellCommand); //命令行的第一项是 bash 或 cmd /c
> if (!scriptPath.isEmpty()) {
>+ vargs.add(Shell.WINDOWS ?ExecBatScripStringtPath : ExecShellStringPath); //加上路径
> }
> vargs.add(shellArgs); //加上命令行参数
> vargs.add("1>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stdout");
> vargs.add("2>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stderr");
> StringBuilder command = new StringBuilder();
> for (CharSequence str : vargs) {
>+ command.append(str).append(" "); //构建命令行
> }
> List<String> commands = new ArrayList<String>(); //构建脚本,但是实际上只有一行命令
> commands.add(command.toString()); //把命令行放在脚本中
> ContainerLaunchContext ctx = ContainerLaunchContext.newInstance(
        localResources, shellEnv, commands, null, allTbkens.duplicate(), null);
        //构建 ContainerLaunchContext,即 CLC,包括脚本在内的所有信息都在 CLC 中
> containerListener.addContainer(container.getId(), container);
> nmClientAsync.startContainerAsync(container, ctx); //启动容器

```

至于投运的具体过程,则与 MapReduce 任务的投运大同小异,只是具体的命令行不同而已,这里就不细说了。

AM 也要跟下面打交道,所谓下面,就是由这个 AM 所管辖的任务在运行的那些节点。AM 中也有一个 NMClientAsyncImpl 线程负责处理来自 NodeManager 的事件报告,对于到来的每个有关容器的事件,它都要创建一个 ContainerEventProcessor 线程专门加以处理:

```

class ContainerEventProcessor implements Runnable {}
] ContainerEventProcessor(ContainerEvent event) //构造函数
    > this.event = event
] run()
    > ContainerId containerId = event.getContainerId()
    > if (event.getType() == ContainerEventType.QUERY_CONTAINER) {
    >+ containerStatus = client.getContainerStatus(containerId, event.getNodeId())
    >+ callbackHandler.onContainerStatusReceived(containerId, containerStatus)
    > } else {

```

```

>+ StatefulContainer container = containers.get(containerId)
>+ if (container == null) {
>++ LOG.info("Container " + containerId + " is already stopped or failed")
>+ } else {
>++ container.handle(event) == StatefulContainer.handle(event)
>++ if (isCompletelyDone(container)) containers.remove(containerId)
>+ }
> }

```

这里的 callbackHandler 是 NMCallbackHandler:

```

class NMCallbackHandler implements NMClientAsync.CallbackHandler {}
] onContainerStatusReceived(ContainerId containerId, ContainerStatus containerStatus)
] onContainerStarted(ContainerId containerId, Map<String, ByteBuffer> allServiceResponse)
> container = containers.get(containerId)
> applicationMaster.nmClientAsync.getContainerStatusAsync(containerId,
                                                                    container.getNodeId())
> ApplicationMaster.publishContainerStartEvent(applicationMaster.timelineClient, container)

```

可见, Client 绕过了 Hadoop 常规的那套作业提交机制, 也绕开了 MRAppMaster, 而另搞一套, 配合 ApplicationMaster 取代了围绕着 MRAppMaster 的那套机制。它并不提供像 MapReduce 那样现成的计算框架, 而是为开发者留下了自行设计和构筑并行计算模型和系统(可以是数据流的, 也可以不是)的余地, 虽然现在使用者还比较少, 但是其所展现的灵活性令人瞩目。

第10章

MapReduce 框架中的数据流

10.1 数据流和工作流

要深入了解和理解一个系统、一个模块乃至一种机制的内部实现,就应考察其内部的两个“流(flow)”:一个是控制流,就是程序的流程;另一个是数据流,就是所处理的数据从源头到终点之间的流动和变化。在有些系统中数据流是在控制流的作用下形成的,而另一些系统则是因数据的驱动才转动起来的。在同一个系统中,也往往是这两种情况兼有。所以这二者无法也无须截然分开,但是对于数据流的考察肯定可以使我们对目标有一个更全面和完整的认识。

对用户而言,Hadoop 最主要的作用和特征就是 MapReduce,所以对其 MR 框架中数据流的考察,即对于数据从数据源到作为最终产物的输出之间各阶段的流动和改变的考察,就尤为重要,尤有帮助。

从宏观上说,MR 框架主要就是 Map 和 Reduce 这两个阶段。但是实际上远不是那么简单,这两个宏观的阶段都进一步划分成好几个更微观的阶段。以前面提到过的排序(Sort)阶段为例,Mapper 的输出端有个由框架提供的局部排序阶段,而 Reducer 输入端的收取(Fetch)和合并(Merge),以至汇合(Combine)阶段又带有排序的成分,由此而形成的全局排序功能也是由框架提供的,并且往往并不能引起使用者的注意,但是它的作用却真切地存在着。

以 mapreduce-examples 目录下的示例 Sort.java 为例,其 Mapper 就采用框架的默认 Mapper,表面上(除类型转换外)什么事也不干,实际上却利用 MR 框架中固有的 Sort 阶段完成了排序。

反过来,如果不是把 Reducer 的数量设定为 0 而明确跳过 Reduce 阶段,也没有以自己开发的软件来替换代码中的 Shuffle 类,那么 YARN 框架提供的 Sort 阶段就是用户绕不过去的。而这又决定了在 Map 与 Reduce 两个阶段之间严格说来只能有“工作流”而不是“数据流”,因为排序只能在全部数据都到位之后才能进行。打个比方,就是 Mapper 与 Reducer 之间其实不只是一条河流,而是一个水库(严格地说水库也并非把全部的水都蓄满了之后才放水)。数据流与工作流的差别主要在于数据流动的粒度。数据流是小粒度的、均匀的数据流动;而工作流则是最大粒度(包含全部数据)的、成批的数据流动。所以工作流天然就是一种批处理,而且是极端的批处理(全部数据都在同一批次)。

由此可见,这些用户看不见的、更为微观的阶段实际上起着相当重要的作用,而且对整个系统的结构和性质也有很大影响。

Hadoop 的代码中为作业的操作流程(即控制流)定义了几个阶段:

```
enum Phase {
    STARTING, MAP, SHUFFLE, SORT, REDUCE, CLEANUP
}
```

这是控制流意义上的阶段划分,如果前一阶段不结束,后面的阶段就不会开始,所以平时 (Mapper 正在进行处理的时候)在 MAP 与 REDUCE 这两个阶段之间没有数据流动,没有并发。这样的控制流所造成的是“工作流”。

但是,尽管如此,我们还是可以大致上把数据在 MR 框架中的流程称为“数据流”,而且从 MAP 到 SHUFFLE 到 SORT 到 REDUCE 的次序也正好与数据的流动相符。更何况在其中的许多局部,例如 Map 阶段内部,在其输入/处理/输出这几个阶段中还真有小粒度的、均匀的数据流动。

那么数据在 MR 这个框架中的流程究竟是怎样的,里面要经历一些什么样的阶段呢? 我们不妨再从 Mapper 和 Reducer 的控制流入手进行更深入的考察。

先看 Hadoop 代码中 Mapper 这个类的摘要,用户提供的 Mapper 都是对这个类的扩充:

```
class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {}
] abstract class Context
    implements MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {}
] run(Context context)    //context 是作为调用参数传下来的
    > setup(context)
    >> // NOTHING, 空函数
    > while (context.nextKeyValue()) {
    >+ map(context.getCurrentKey(), context.getCurrentValue(), context)
    >+> context.write((KEYOUT) key, (VALUEOUT) value)
    > cleanup(context)
    >> // NOTHING, 空函数
```

我们来看 Mapper.run()。这里的 setup()和 cleanup()都是空的,但是用户当然可以用自己的 setup()和 cleanup()覆盖取代这两个函数。Context 是个抽象类,在这里并没有具体定义,只是说实现了 MapContext 界面,即必须提供 MapContext 界面中规定的那些函数,这里面就包括了 nextKeyValue()、getCurrentKey()、getCurrentValue(),还有 write()这些方法。具体的 Context 是由 MR 框架传给 Mapper 的,不管其内部都有些什么,反正 Mapper 只关心和引用它的这几个方法函数。下面我们将看到,事实上这个 Context 起着关键的作用。从摘要中可见,Mapper 的核心就是一个 while 循环,这个循环能够继续的条件就是 context.nextKeyValue()返回 true,也即数据的输入尚未穷尽。而只要还有下一个 KV 对,则 context.getCurrentKey()和 context.getCurrentValue()自会返回其中的 K 和 V。这里的 map()函数其实没进行什么处理,只是实现了可能需要的对于 K 和 V 的类型转换,就通过 context.write()写出去了。这样也好,可以让我们把注意力更好地集中到这个框架本身。不过当然,这个 map()函数也是可以也应该被覆盖的。于是我们自然要问,“下一个 KV 对”究竟来自何处,是怎么读进来的? 而写出去的时候又写到了哪里? 这数据是怎样到达 Reducer 一方的?

再看 Reducer 的摘要。同样,用户提供的 Reducer 都是对这个类的扩充:

```
class Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT> {}
] abstract class Context
    implements ReduceContext<KEYIN,VALUEIN,KEYOUT,VALUEOUT> {}
] run(Context context)    //context 是作为调用参数传下来的
    > setup(context)
    > while (context.nextKey()) {
    >+ values = context.getValues() == ReduceContextImpl.getValues()
    >+> return iterable == ValueIterable()
    >+ reduce(context.getCurrentKey(), values, context)
    >+> for(VALUEIN value: values) {
    >+>+ context.write((KEYOUT) key, (VALUEOUT) value)
    >+> }
    >+ ...
    > }
    > cleanup(context)
```

这里的 setup()、cleanup(),特别是 Context 都与 Mapper 中的相似,但请注意,MR 框架传给 Reducer 的与传给 Mapper 的 Context 是不一样的,这个 Context 所实现的是 ReduceContext 界面,它提供的是 nextKey()、getValues()和 write()。当然,虽然都有个 write(),但二者的方法和目标都不一样。

如前所述,Reducer 的输入是 [K, V, V, V, ...] 这样的形式,而 Mapper 的输出却只是一个一个的 KV 对。这中间显然存在着某种处理,把 K 相同的众多 V 归并到了一起。而要做这样的归并,最合理的方法就是先做排序再做合并。而且,鉴于 Reducer 与 Mapper 通常不在同一节点上,这一步似乎应该在 Mapper 这一边完成,这样可以减少网络上的流量。然后,由于 Mapper 与 Reduce 常常是多对一的关系,来自多个 Mapper 的数据在 Reducer 这一边又有如何合并的问题,在此过程中也可能有排序的需要:既然来自多个 Mapper 的数据都是排好序的,那么在 Mapper 与 Reducer 之间就自然可以形成一种 MergeSort 的排序机制。

当然,Mapper 的输入和 Reducer 的输出也需要加以考察。后面我们会看到,其中 Mapper 的输入尤为复杂和重要。

这样,如果我们从 MR 框架的数据来源开始,考察数据怎样一步步流过这个框架中的各个阶段,经历了一些什么变化,最后怎么从框架流出,我们就对整个 MR 框架有了比较深入的理解。

10.2 Mapper 的输入

我们从数据源开始。MR 的数据一般来源于 HDFS 文件,但是也可以来源于例如查询数据库的输出,也可以来源于某种数据“生成器”。当然,原始的文件最初只存在于某个宿主机文件系统中,但是那离 HDFS 文件只是一步之遥,只要拷贝一下即可。文件也没有规定必须是某种特定格式的文件,例如也可以是网页。但是,对于具体的数据源,MR 框架显然必须能从

中读出数据,形成 KV 对,并将其作为 nextKeyValue()调用的输出,由 Mapper 用作调用map()的参数,相当于将其馈送给 map()的输入。

这就是 InputFormat 的作用。有什么样的数据源,什么样的数据格式,就得采用什么样的 InputFormat。我们在前面看到,具体的应用在提交作业前都要调用 job.setInputFormatClass(),就是这个道理。Hadoop 定义了一个抽象类 InputFormat,规定每一种输入格式都必须自行提供 getSplits()和 createRecordReader()这两个函数:

```
abstract class InputFormat<K, V> {}
] abstract getSplits()
] abstract createRecordReader()
```

其中 getSplits()应返回一个 List<InputSplit>,即一个 Mapper 输入“片(Split)”的列表。这是因为 Mapper 一般都有多份,而输入源,姑且假定其为文件,却只有一个,所以得要把该文件,或者说所有输入数据的集合,分成许多片,一个片对应着一个 Mapper。如果给定 Mapper 的数量,那么片的数量也就随之确定了。反过来,如果不给定 Mapper 的数量,则怎样分片比较合适就为确定 Mapper 的数量提供了依据。但是片该怎么分,却大有讲究。例如 HDFS 文件,本身就是按“块”存储的,那么以一个块作为一个片,然后把相应的 Mapper 安排在这个块所在的节点上,就比较合适。或者,如果觉得块(默认 64MB)还是太大,那也可以把块再切分成几个片,但是跨块分片总是不太合适。然而,如果数据源不是文件,而是数据库查询的实时输出,那就又不同了。所以,对于如何分片,具体的 InputFormat 是最有“话语权”的。至于 createRecordReader(),则顾名思义是创建 RecordReader。每种 InputFormat 都有配套的 RecordReader,具体的“记录读入器”负责从数据源的指定 Split 中读入“记录”,形成 KV 对,供指定的 Mapper 使用。Mapper 中对于 context.nextKeyValue()的调用,最后就落实到 RecordReader 的头上。每个 RecordReader 都必须提供 nextKeyValue()、getCurrentKey()、getCurrentValue()三个函数。

事实上,Hadoop 提供了很多具体的 InputFormat 类型,都是对抽象类 InputFormat 的直接或间接的扩充,例如:

```
abstract class FileInputFormat<K, V> extends InputFormat<K, V> {}

class FixedLengthInputFormat extends FileInputFormat<LongWritable, BytesWritable>
    implements JobConfigurable {}

class KeyValueTextInputFormat extends FileInputFormat<Text, Text>
    implements JobConfigurable {}

class StreamInputFormat extends KeyValueTextInputFormat {}

class SequenceFileInputFormat<K, V> extends FileInputFormat<K, V> {}

class SequenceFileAsBinaryInputFormat extends
    SequenceFileInputFormat<BytesWritable, BytesWritable> {}

class SequenceFileAsTextInputFormat extends SequenceFileInputFormat<Text, Text> {}

class SequenceFileInputFilter<K, V> extends SequenceFileInputFormat<K, V> {}

class TextInputFormat extends FileInputFormat<LongWritable, Text>
```

```

        implements JobConfigurable {}

class NLineInputFormat extends FileInputFormat<LongWritable, Text>
        implements JobConfigurable {}

class AutoInputFormat extends FileInputFormat {}

abstract class CombineFileInputFormat<K, V> extends FileInputFormat<K, V> {}

class CombineSequenceFileInputFormat<K, V> extends CombineFileInputFormat<K, V> {}
class CombineTextInputFormat extends CombineFileInputFormat<LongWritable, Text> {}

```

这里面 `KeyValueTextInputFormat` 是指输入文件的每一行本身就是一个由逗号分隔的 KV 对,而不是说只有这个 `InputFormat` 才产生 KV 对。相应地,与之配套的 `RecordReader` 就是 `KeyValueLineRecordReader`。

实际上, `TextInputFormat` 和 `SequenceFileInputFormat` 都是很常用的输入格式,相应的 `RecordReader` 为 `LineRecordReader` 和 `SequenceFileRecordReader`。

这里还有个 `SequenceFileInputFilter`,是个带有过滤功能的 `RecordReader`,可以对输入数据实行基于正则表达式(Regex)、百分比、MD5 等条件的过滤。与之配套的是 `FilterRecordReader`,那是对 `SequenceFileRecordReader` 的扩充。其实 `FilterRecordReader.nextKeyValue()` 并不直接从数据源读入,它只是调用 `SequenceFileRecordReader.nextKeyValue()` 获取输入,然后加以过滤。

如前所述,数据源也可以是数据库的输出,那就要有适合数据库的 `InputFormat`:

```

class DBInputFormat<T extends DBWritable> extends InputFormat<LongWritable, T>
        implements Configurable {}

class DataDrivenDBInputFormat<T extends DBWritable> extends DBInputFormat<T>
        implements Configurable {}

```

这两种以数据库输出为数据源的 `InputFormat` 代表着两种操作方式。前者是这样的:在 `getSplits` 时对数据库进行一次查询,看看输出有多大,据以决定如何分片。然后,在首次调用 `nextKeyValue()` 时又向数据库发出一个查询,将数据库的输出存放在一个缓冲文件中,再按预定的分片读入。配套的 `RecordReader` 则按实际使用数据库的不同,可以是 Hadoop 提供的 `OracleDBRecordReader`、`MySQLDBRecordReader` 或一般的 `DBRecordReader`。之所以如此,是因为这些数据库的 SQL 语法略有不同。Hadoop 代码中的示例 `DBCountPageView.java`,就是用来演示这种方式的。显然,这跟手工做一次查询把输出放在中间文件中,然后以中间文件为数据源的方法并无原则上的不同,在数据库与 MR 之间并没有形成数据流,而只是工作流。有鉴于此, Hadoop 又提供了一种更接近于数据流的 `DataDrivenDBInputFormat`。所谓 `DataDriven`,就是用来强调表示这是“数据驱动”的,即数据流模式的输入。与之配套的 `RecordReader` 是 `MySQLDataDrivenDBRecordReader`(专用),或 `DataDrivenDBRecordReader`(通用)。这种方式是这样:先大致了解一下情况,然后由每个 `RecordReader`,实际上就是每个 Mapper,各自发起一个查询,在 `SELECT` 语句中通过 `WHERE` 子句给定不同的条件,比方说 `RowId` 大于等于多少并且小于多少。这样,就可以使每个 Mapper 的 `RecordReader` 都获得数据库的一部分输出,并且可以形成真正的数据流(数据库边输出 Mapper 边处理),而无须使用

中间文件,而且形成了数据库与 Mapper 之间的并行或并发。

至于并非来自某个数据源,而是由程序自己生成的输入,则 Hadoop 代码中的示例 `QuasiMonteCarlo.java` 是个很好的例子。这个示例采用准蒙特卡罗方法计算圆周率 π , 在一个单位正方形中随机生成很多点的坐标,计算这些点是否落在这个正方形的内切圆内或圆周上,这样可以计算落在圆内或圆外的概率,从而就能算出 π 的值。那么,这些点的坐标是怎么生成的呢,可以直接由 `map()` 函数自己生成,连 `RecordReader` 也可不用,`map()` 内部有个 `for` 循环,生成一个点就计算一下,显然这样效率最高。当然,设计用一个特殊的 `RecordReader` 来随机生成这些点的坐标也是可以的。

下面我们以 Hadoop 源码中的示例 `WordCount` 为例说明 Mapper 输入端的那一段数据流。

```
WordCount.main()
> ...
> job.setMapperClass(TokenizerMapper.class)
> ...
> for (int i = 0; i < otherArgs.length - 1; ++ i) {
>+ FileInputFormat.addInputPath(job, new Path(otherArgs[i])) //可以有多个输入文件
> }
> FileOutputFormat.setOutputPath(job, new Path(otherArgs[otherArgs.length - 1]))
> job.waitForCompletion(true)
```

这个 App 在准备提交的“任务书”中没有设置 `InputFormat`,那就默认为普通的字符文件,即 `TextInputFormat`。

注意,这里也没有设置 Mapper 的数量,Mapper 的数量是由系统根据输入文件和分片的情况决定的。

Mapper 输入端的复杂之处,除输入数据源的多样性外,还来自输入分片(Split)。一般 Mapper 有很多个,而数据源,特别是输入文件却只有一个,但通常很大。在 HDFS 文件系统中,文件分块存储在多个节点上,但逻辑上仍是一个完整的文件。很多个 Mapper 分头处理同一个文件,那当然就得把文件分拆成“片”,每个 Mapper 一片。这样,每个 Mapper 的输入就来自同一个文件的不同部位或同一数据源的不同视野(view)。

要搞清输入分片的问题,需要回溯到提交作业的时候,把来龙去脉搞清楚。这里我们对此一方面重温,一方面要做进一步深入和细化的考察。

作业提交流程中关键的一步是 `submitJobInternal()`,我们就从这里开始。不过这次我们只拣与 Mapper 输入有关的内容做摘要,别的信息都视作无关紧要而忽略不计了。

我们知道, `JobSubmitter` 在提交作业时发送给 RM 的是一个类似于“作业单”那样的 `ApplicationSubmissionContext`。但是其实那也并非全部,许多信息是以文件的形式传递的,放在“作业单”中的只是文件路径。这方面最典型的就是输入数据文件本身,但还有别的文件,其中就有 `SplitMetaFile` 和 `SplitFile`,那是对于输入分片的描述。`JobSubmitter` 在 `submitJobInternal()` 通过 `writeSplits()` 生成这两个文件,把文件路径写在“作业单”中;以后 `MRAppMaster` 就在其资源本地化的过程中把这两个文件拷贝过去。我们在第 5 章讲作业提交时做过 `writeSplits()` 的摘要,为免来回翻阅,这里再做个角度略有不同的摘要以方便阅读:


```
[Job.submit() > JobSubmitter.submitJobInternal() > writeSplits()]

writeSplits(org.apache.hadoop.mapreduce.JobContext job, Path jobSubmitDir)
> JobConf jConf = (JobConf) job.getConfiguration()
> if (jConf.getUseNewMapper()) {
>+ int maps = writeNewSplits(job, jobSubmitDir) //Split 的数量决定了 Mapper 的数量
>+ conf = job.getConfiguration()
>+ clazz = job.getInputFormatClass() == JobContextImpl.getInputFormatClass()
>+> return (conf.getClass(INPUT_FORMAT_CLASS_ATTR, TextInputFormat.class)
//获取本作业的输入格式,默认 TextInputFormat
>+> InputFormat<?,?> input = ReflectionUtils.newInstance(clazz, conf) //创建该类对象
>+> List<InputSplit> splits = input.getSplits(job) == TextInputFormat.getSplits(job)
== FileInputFormat.getSplits(job) // TextInputFormat 继承 FileInputFormat
>+> T[] array = (T[]) splits.toArray(new InputSplit[splits.size()]) //将 List 转化成数组
>+> Arrays.sort(array, new SplitComparator()) //对数组中的 Split 按大小排序,大的在前
>+> JobSplitWriter.createSplitFiles(jobSubmitDir, conf,
jobSubmitDir.getFileSystem(conf), array)
>+>> splfile = JobSubmissionFiles.getJobSplitFile(jobSubmitDir)
>+>>> return new Path(jobSubmissionDir, "job.split")
>+>> FSDataOutputStream out = createFile(fs, splfile, conf)
>+>> SplitMetaInfo[] info = JobSplitWriter.writeNewSplits(conf, splits, out)
//将分片信息写入 job.split 文件,这里的 splits 就是上面的 array
-----
>+>> metafile = JobSubmissionFiles.getJobSplitMetaFile(jobSubmitDir)
>+>>> return new Path(jobSubmissionDir, "job.splitmetainfo")
>+>> perm = new FsPermission(JobSubmissionFiles.JOB_FILE_PERMISSION)
>+>> writeJobSplitMetaInfo(fs, metafile, perm, splitVersion, info)
>+> return array.length
> } else {
>+ maps = writeOldSplits(jConf, jobSubmitDir)
> }
> return maps //这是应有的 Mapper 数量,有几个 Split 就应该有几个 Mapper
```

这里所谓的 Split 是 InputSplit 对象。不过 InputSplit 是抽象类,具体扩充落实 InputSplit 的类则有好多个,因为不同数据源的分片是不一样的。就数据文件而言,它的 InputSplit 是 FileSplit,其数据部分的摘要如下:

```
class FileSplit extends InputSplit implements Writable {}
] Path file //文件路径
] long start //本 Split 在文件中的起点
] long length //本 Split 的长度
```

```

] String[] hosts          //本 Split 内容所在的节点,一个 Split 可能涉及多个节点
] SplitLocationInfo[] hostInfos //所涉及节点的信息,例如是在内存中还是在磁盘上

```

这样,根据一个分片的 FileSplit 对象,就可以知道这个分片的数据在什么地方。

输入文件的分片是通过具体输入格式的 getSplits()方法完成的。输入文件可以只有一个,也可以有多个, getSplits()会将每个输入文件分成片,最后返回一个 List。对一个输入文件进行分片时, getSplits()通过 computeSplitSize()计算每个分片的大小,计算的依据有具体文件系统中块(Block)的大小(HDFS 中块的大小为 64MB 或 128MB),配置中的最小片长度和最大片长度。限于篇幅, getSplits()的代码就留给读者自行阅读分析了。

需要强调的是,并不是根据文件的大小和预定的 Mapper 数量确定片的大小,而是根据文件的大小和片的大小确定片的数量。而片的数量一旦确定, Mapper 的数量也就确定了。

另一个文件是 JobSplitMetaFile,其内容是每个 Split 数据文件的路径:

```

[submitJobInternal() > writeSplits()>writeJobSplitMetaInfo()]

JobSplitWriter.writeJobSplitMetaInfo(FileSystem fs, Path filename, FsPermission p,
int splitMetaInfoVersion, JobSplit.SplitMetaInfo[] allSplitMetaInfo)
> FSDataOutputStream out = FileSystem.create(fs, filename, p) //创建文件
> out.write(JobSplit.META_SPLIT_FILE_HEADER) //前面是文件头
> WritableUtils.writeVInt(out, splitMetaInfoVersion)
> WritableUtils.writeVInt(out, allSplitMetaInfo.length)
> for (JobSplit.SplitMetaInfo splitMetaInfo : allSplitMetaInfo) {
>+ splitMetaInfo.write(out) == JobSplit.SplitMetaInfo.write(out)
>+> WritableUtils.writeVInt(out, locations.length) //这是 SplitMetaInfo 中 locations[] 的长度
>+> for (int i=0; i < locations.length; i++) { //对于 locations[] 中的每个 Split 文件:
>+>+ Text.writeString(out, locations[i]) //将其文件路径记入 JobSplitMetaFile
>+> }
>+> WritableUtils.writeVLong(out, startOffset)
>+> WritableUtils.writeVLong(out, inputDataLength)
> }

```

这里的数组 SplitMetaInfo.locations[] 来自 writeSplits() > writeNewSplits() 中对 getSplits()的调用。本书第 5 章在讲述作业提交过程时在 writeSplits()的摘要中展开过对于 getSplits()的调用,读者可以回过去重温一下。

当作业被提交到 RM 时, RM 并不关心这个作业的输入如何分片,它关心的是要找一个节点去建立这个作业的 MRAppMaster。所以关心 Split 文件的是 MRAppMaster。

MRAppMaster 首先会在它本地创建这个作业的 JobImpl 对象,然后在完成一些初始化操作之后就向此 JobImpl 对象发出 JOB_INIT 事件,使其执行 InitTransition.transition()。此时上述两个有关输入分片的文件已通过资源本地化拷贝到了本地。我们就从这次跳变看起:

```

JobImpl.InitTransition.transition(JobImpl job, JobEvent event)
> TaskSplitMetaInfo[] taskSplitMetaInfo = createSplits(job, job.jobId) //处理输入分片信息

```

```

>>> TaskSplitMetaInfo[] allTaskSplitMetaInfo = SplitMetaInfoReader.readSplitMetaInfo(
                                job.oldJobId, job.fs, job.conf, job.remoteJobSubmitDir)
>>>> Path metaSplitFile = JobSubmissionFiles.getJobSplitMetaFile(jobSubmitDir)
>>>> String jobSplitFile = JobSubmissionFiles.getJobSplitFile(jobSubmitDir).toString()
>>>> FSDataInputStream in = fs.open(metaSplitFile)
>>>> ...
>>>> int numSplits = WritableUtils.readVInt(in)
                                //从 meta 文件读取 Split 的数量,这也决定了该有几个 MapTask
>>>> JobSplit.TaskSplitMetaInfo[] allSplitMetaInfo = new JobSplit.TaskSplitMetaInfo[numSplits]
>>>> for (int i = 0; i < numSplits; i++) {
>>>>+ JobSplit.SplitMetaInfo splitMetaInfo = new JobSplit.SplitMetaInfo()
>>>>+ splitMetaInfo.readFields(in)
>>>>+ JobSplit.TaskSplitIndex splitIndex =
                                new JobSplit.TaskSplitIndex(jobSplitFile, splitMetaInfo.getStartOffset())
                                //为 Split 文件的每个片都创建一个 TaskSplitIndex 对象
>>>>+ allSplitMetaInfo[i] = new JobSplit.TaskSplitMetaInfo(splitIndex,
                                splitMetaInfo.getLocations(), splitMetaInfo.getInputDataLength())
                                //这个数组中的元素是 TaskSplitMetaInfo 对象,TaskSplitIndex 是其成分之一
>>>> } //end for
>>>> return allSplitMetaInfo
>>> return allTaskSplitMetaInfo
> job.numMapTasks = taskSplitMetaInfo.length //有几个 Split 就该有几个 MapTask
> job.numReduceTasks = job.conf.getInt(MRJobConfig.NUM_REDUCES, 0)
                                //ReduceTask 的数量则是预设的,默认 0
> createMapTasks(job, inputLength, taskSplitMetaInfo)
>>> for (int i = 0; i < job.numMapTasks; ++i) { //逐个创建 MapTask,它们的 Split 各不相同
>>>+ task = new MapTaskImpl(job.jobId, i, job.eventHandler, job.remoteJobConfFile,
                                job.conf, splits[i], job.taskAttemptListener, job.jobToken, job.jobCredentials,
                                job.clock, job.applicationAttemptId.getAttemptId(),
                                job.metrics, job.appContext)
>>>+ job.addTask(task)
>>> }

```

在此之前,两个有关输入分片的文件已在资源本地化的过程中被复制过来,现在就从中恢复出分片信息,并根据分片的数量创建出相应的 MapTaskImpl 对象,每个 MapTaskImpl 所使用的输入数据片 splits[i]是不一样的。这些 MapTaskImpl 将被投送到不同的节点上,而每个具体数据片所在的节点正是它们最佳的投送目的地。注意,MapTaskImpl 是对 TaskImpl 的扩充,每个 MapTaskImpl 对象同时也是 TaskImpl 对象。

然后,当 JobImpl 完成了准备,在 scheduleTasks() 中向各个 TaskImpl 发出 T_SCHEDULE 事件时,每个 TaskImpl 对象的状态机会发生一次 InitialScheduleTransition 跳变,执行 InitialScheduleTransition.transition(),在那里调用 addAndScheduleAttempt()。后者则通过

addAttempt() 创建 TaskAttempt 对象。对于 MapTaskImpl 而言, 所创建的是 MapTaskAttemptImpl。

MapTaskAttemptImpl 的创建是这样的:

```
[TaskImpl.addAttempt() > MapTaskImpl.createAttempt()]
```

```
MapTaskImpl.createAttempt()
```

```
> return new MapTaskAttemptImpl(getID(), nextAttemptNumber, eventHandler,
    jobFile, partition, taskSplitMetaInfo, conf, taskAttemptListener,
    jobToken, credentials, clock, appContext)
```

注意这里创建 MapTaskAttemptImpl 对象时的实参有 partition 和 taskSplitMetaInfo。前者告诉 Mapper 有几个 Reducer, 因而应该将其输出分成多少个 Partition; 后者则说明这个 Mapper 的输入是哪一个 Split, 在哪一个 HDFS 文件中, 相应的块在哪一个或哪几个节点上。可见, 所创建的 MapTaskAttemptImpl 对象中将保有这些信息。就这样, 有关 Split 的信息就一层层下传。

到了在某个目标节点上创建 MapTask 的时候, 这些信息也照样都传了下来, 直到 MapTask.run() 用上了这些信息。

```
[YarnChild.main() > PrivilegedExceptionAction.run() > taskFinal.run()]
```

```
MapTask.run(JobConf job, TaskUmbilicalProtocol umbilical)
```

```
> initialize(job, getJobID(), reporter, useNewApi)
```

```
>> jobContext = new JobContextImpl(job, id, reporter)
```

```
>> taskContext = new TaskAttemptContextImpl(job, taskId, reporter)
```

```
>> outputFormat = ReflectionUtils.newInstance(taskContext.getOutputFormatClass(), job)
```

```
>> committer = outputFormat.getOutputCommitter(taskContext)
```

```
>> Path outputPath = FileOutputFormat.getOutputPath(conf)
```

```
>> if (outputPath != null) {
```

```
>>+ if ((committer instanceof FileOutputCommitter)) {
```

```
>>++ FileOutputFormat.setWorkOutputPath(conf,
```

```
    ((FileOutputCommitter) committer).getTaskAttemptPath(taskContext))
```

```
>>+ } else {
```

```
>>++ FileOutputFormat.setWorkOutputPath(conf, outputPath)
```

```
>>+ }
```

```
>> }
```

```
>> committer.setupTask(taskContext)
```

```
>> Class <? extends ResourceCalculatorProcessTree> clazz =
```

```
    conf.getClass(MRConfig.RESOURCE_CALCULATOR_PROCESS_TREE, null,
```

```
    ResourceCalculatorProcessTree.class)
```

```
>> pTree = ResourceCalculatorProcessTree.getResourceCalculatorProcessTree(
```

```
    System.getenv().get("JVM_PID"), clazz, conf)
```

```
>> if (pTree != null) {
```

```

>>>+ pTree.updateProcessTree()
>>>+ initCpuCumulativeTime = pTree.getCumulativeCpuTime()
>>> }
> runNewMapper(job, splitMetaInfo, umbilical, reporter)

```

最后, MapTask.run()调用 runNewMapper(),也把这些信息作为参数传了下来:

```
[YarnChild.main() > PrivilegedExceptionAction.run() > MapTask.run() > runNewMapper()]
```

```

runNewMapper(JobConf job, TaskSplitIndex splitIndex, ...)
> taskContext = new TaskAttemptContextImpl(job, getTaskID(), reporter)
    //TaskAttemptContextImpl 是对 JobContextImpl 的扩充
> clazz = taskContext.getInputFormatClass() == JobContextImpl.getInputFormatClass()
>>> return conf.getClass(INPUT_FORMAT_CLASS_ATTR, TextInputFormat.class)
    //未加设定就默认 TextInputFormat
> inputFormat = ReflectionUtils.newInstance(clazz, job) //见上
> path = new Path(splitIndex.getSplitLocation())
> offset = splitIndex.getStartOffset()
> InputSplit split = getSplitDetails(path, offset) //恢复输入数据片的 InputSplit 对象
> input = new NewTrackingRecordReader<INKEY, INVALUE> (split, inputFormat,
    reporter, taskContext)
    //创建与具体 InputFormat 相称的 RecordReader
>>> if (split instanceof org.apache.hadoop.mapreduce.lib.input.FileSplit) {
>>>+ matchedStats = getFsStatistics(((FileSplit) split).getPath(), taskContext.getConfiguration())
>>> }
>>> fsStats = matchedStats
>>> this.real = inputFormat.createRecordReader(split, taskContext)
    == TextInputFormat.createRecordReader(InputSplit split, TaskAttemptContext context)
>>>> String delimiter = context.getConfiguration().get("textinputformat.record.delimiter")
    //单词分隔符,例如空格和标点符号
>>>> if (null != delimiter)
    byte[] recordDelimiterBytes = delimiter.getBytes(Charsets.UTF_8)
>>>> return new LineRecordReader(recordDelimiterBytes)
    // TextInputFormat 的 RecordReader 是 LineRecordReader
>>>>> this.recordDelimiterBytes = recordDelimiter
> ...
> mapContext = new MapContextImpl<INKEY, INVALUE, OUTKEY, OUTVALUE>(
    job, getTaskID(), input, output, committer, reporter, split)
    //创建用于 Mapper 的 Context
> wrappedmapper = new WrappedMapper<INKEY, INVALUE, OUTKEY, OUTVALUE>()
> mapperContext = wrappedmapper.getMapContext(mapContext)

```

```

>>> return new Context(mapContext) //这是定义于 WrappedMapper 内部的 Context
>>>> this.mapContext = mapContext
//所以 WrappedMapper.Context.mapContext 是个 MapContextImpl
> input.initialize(split, mapperContext) //数据输入源的初始化
== NewTrackingRecordReader.initialize(split, mapperContext)
>>> real.initialize(split, context) == LineRecordReader.initialize(split, context)
>>>> FileSplit split = (FileSplit) genericSplit
>>>> Configuration job = context.getConfiguration()
>>>> start = split.getStart()
>>>> end = start + split.getLength()
>>>> Path file = split.getPath()
>>>> FileSystem fs = file.getFileSystem(job)
>>>> fileIn = fs.open(file)
>>>> CompressionCodec codec = new CompressionCodecFactory(job).getCodec(file)
>>>> if (null != codec) {
>>>>+ ...
>>>> } else {
>>>>+ fileIn.seek(start)
>>>>+ in = new SplitLineReader(fileIn, job, this.recordDelimiterBytes)
>>>>+ filePosition = fileIn
>>>> }
> mapper.run(mapperContext) == Mapper.run()
>>> setup(context)
>>> while (context.nextKeyValue()) {
>>>+ map(context.getCurrentKey(), context.getCurrentValue(), context)
>>> }
>>> cleanup(context)
> ...

```

我们现在考察的是 Mapper 的输入,所以不必关心具体的 map() 函数是什么样的,而只需要知道调用 map() 函数时的 KV 对是怎么来的就行了。

显然,这个 KV 对是通过 context.nextKeyValue() 读进来的,这里的 context 是个 MapContextImpl 对象。

```
[MapTask.run() > runNewMapper() > MapContextImpl.nextKeyValue()]
```

```
MapContextImpl.nextKeyValue()
```

```

> return reader.nextKeyValue() == NewTrackingRecordReader.nextKeyValue()
>>> boolean result = real.nextKeyValue() == LineRecordReader.nextKeyValue()
>>>> if (key == null) key = new LongWritable()
>>>> key.set(pos)

```



```

>>> if (value == null) value = new Text()
>>> while (getPosition() <= end || in.needAdditionalRecordAfterSplit()) {
>>>+ if (pos == 0) {
>>>++ newSize = skipUtfByteOrderMark()
>>>+ } else {
>>>++ newSize = in.readLine(value, maxLineLength, maxBytesToConsume(pos))
>>>+               == SplitLineReader.readLine(...) == LineReader.readLine()
>>>+               //SplitLineReader 是对 LineReader 的扩充
>>>++> if (this.recordDelimiterBytes != null) {
>>>++>+ return readCustomLine(str, maxLineLength, maxBytesToConsume)
>>>++> } else {
>>>++>+ return readDefaultLine(str, maxLineLength, maxBytesToConsume)
>>>++> }
>>>++ pos += newSize
>>>+ }
>>>+ if ((newSize == 0) || (newSize < maxLineLength)) break
>>> } //end while
>>> if (newSize == 0) {
>>>+ key = value = null
>>>+ return false //使 mapper.run() 中的 while 语句停止循环
>>> } else {
>>>+ return true //使 mapper.run() 中的 while 语句继续循环
>>> }
>> return result //返回 true 或者 false,使 mapper.run() 中的 while 语句继续或停止循环

```

这样,Mapper 的输入端就或因读到了一个 KV 对而继续循环,或因输入数据已经穷尽而停止循环。而 map() 函数具体进行什么计算,我们就不关心了。以前讲过,像这样的计算就是 Lambda 演算。

文件的格式可以不同,数据源的性质也可以不同(例如数据库查询输出),但原理都是一样的。从数据流动的观点看,从数据源到 Mapper 是一个均匀而连续的数据流,数据的粒度是 KV 对。

10.3 Mapper 的输出缓冲区 MapOutputBuffer

明白了数据怎么从其源头到达 map() 的输入端(作为调用参数),我们再看 map() 的输出,就是代码中 context.write() 这个语句的内涵,这就要复杂多了。

首先我们得搞清 Mapper 作为调用参数传给 map() 的 context 究竟是什么 context。我们看到,这是在 Mapper.run() 受到调用的时候作为参数传下来的。而调用 Mapper.run() 的则是 MapTask.runNewMapper()。前面我们看过 MapTask.runNewMapper() 的代码摘要,现在结合这个 context 的来源再重温一下,看得再深入一些:

```
[MapTask.run() > runNewMapper()]
```

```
runNewMapper(JobConf job, TaskSplitIndex splitIndex,
              TaskUmbilicalProtocol umbilical, TaskReporter reporter)
> taskContext = new TaskAttemptContextImpl(job, getTaskID(), reporter)
> mapper = ReflectionUtils.newInstance(taskContext.getMapperClass(), job)
              //确定该用哪一种具体的 Mapper,然后创建
> inputFormat = ReflectionUtils.newInstance(taskContext.getInputFormatClass(), job)
> split = getSplitDetails(new Path(splitIndex.getSplitLocation()), splitIndex.getStartOffset())
              //确定这个 Mapper 所用的输入是哪一个 Split
> input = new NewTrackingRecordReader<INKEY, INVALUE>(split,
              inputFormat, reporter, taskContext)
              //创建与具体 InputFormat 相称的 RecordReader
> if (job.getNumReduceTasks() == 0) //如果 Reducer 的数量设置为 0,就直接输出
>+ output = new NewDirectOutputCollector(taskContext, job, umbilical, reporter)
              //创建直接输出的 Collector
> else //要不然就创建通往 Reducer 的 Collector
>+ output = new NewOutputCollector(taskContext, job, umbilical, reporter)
              //创建 Mapper 的输出 RecordWriter,包括 collector 和 partitioner
>+> collector = createSortingCollector(job, reporter) //创建通向排序阶段的 Collector
>+> partitions = jobContext.getNumReduceTasks() //有几个 Reducer,就有几个 partition
>+> if (partitions > 1) { //如果有多个 partition
>+>+ clazz = jobContext.getPartitionerClass()
              //应该是对抽象类 Partitioner 的某种扩充,如果未加设定就默认 HashPartitioner
>+>+ partitioner = ReflectionUtils.newInstance(clazz, job) //创建用户设置的 Partitioner
>+> } else { //只有一个 partition,就动态扩充抽象类 Partitioner
>+>+ partitioner = new org.apache.hadoop.mapreduce.Partitioner<K,V>() {}
              ] getPartition(K key, V value, int numPartitions)
              > return partitions - 1
>+> }
> mapContext = new MapContextImpl<INKEY, INVALUE, OUTKEY, OUTVALUE>(
              job, getTaskID(), input, output, committer, reporter, split)
              //创建用于 Mapper 的 Context
> wrappedmapper = new WrappedMapper<INKEY, INVALUE, OUTKEY, OUTVALUE>()
> mapperContext = wrappedmapper.getMapContext(mapContext)
>> return new Context(mapContext) //这是定义于 WrappedMapper 内部的 Context
>>> this.mapContext = mapContext
              //所以 WrappedMapper.Context.mapContext 是个 MapContextImpl
> input.initialize(split, mapperContext)
> mapper.run(mapperContext)
```

```

> mapPhase.complete()
> setPhase(TaskStatus.Phase.SORT) //进入排序阶段
> statusUpdate(umbilical)
> input.close() //此后 Mapper 不再需要 input
> output.close(mapperContext) //此后 Mapper 不再有 output

```

前面跟输入格式和分片有关的就无须细说了,但是这里要补充说明一下 NewTrackingRecordReader。我们已经知道,有了 InputFormat 之后,就要创建与之相应的 RecordReader。但是这里在具体 RecordReader 外面又包上了一层 NewTrackingRecordReader。不同之处在于 Tracking,就是说对于 RecordReader 的行动是可以跟踪的,所以这里创建时有个参数是 reporter,这个 reporter 就是用来向上级报告的,而具体的 RecordReader 本身则没有此项功能。

跟输出有关的主要是 collector,这是输出数据的“收集器”,context.write()最后就通过 RecordWriter 落实到 collector.collect()。显然,RecordWriter 与 RecordReader 属于同一层次。Hadoop 代码中定义了一个抽象类 RecordWriter(在 mapred 分支上还有个 interface 也叫 RecordWriter),具体 RecordWriter 都是对此抽象类的扩充,其中用于 MapTask 的就是 NewDirectOutputCollector 和 NewOutputCollector。

```

class NewDirectOutputCollector<K,V>
    extends org.apache.hadoop.mapreduce.RecordWriter<K,V> {}
class NewOutputCollector<K,V>
    extends org.apache.hadoop.mapreduce.RecordWriter<K,V> {}

```

这二者都名曰 OutputCollector,实际上却都是 RecordWriter,都是对抽象类 RecordWriter 的扩充。所谓 Collector 只是个名称,只是一种语意上的描述。从上游 Mapper 的角度看这是 Write,是写出;从框架或下游的角度看则是 Collect,是收集。

如果 Reducer 的数量设置为 0,那就没有 Reducer,Mapper 的输出就是整个 MR 框架的输出,这时候用的 RecordWriter 是 NewDirectOutputCollector,情况比较简单。反之,如果至少有一个 Reducer,那么这时候用的 RecordWriter 就是 NewOutputCollector。我们在这里关心的是后者。NewOutputCollector 是定义于 MapTask 内部的一个类:

```

class NewOutputCollector<K,V> extends...hadoop.mapreduce.RecordWriter<K,V> {}
] MapOutputCollector<K,V> collector //实现 MapOutputCollector 界面
] org.apache.hadoop.mapreduce.Partitioner<K,V> partitioner //负责 Mapper 输出的分区
] int partitions //分发目标的个数,即 Reducer 的个数
] NewOutputCollector() //构造函数
] write(K key, V value) //RecordWriter 只写不读
] close(TaskAttemptContext context)

```

NewOutputCollector 中有两个成分都很重要,一个是 collector,还有一个是 partitioner。前者担负实际收集 Mapper 输出并将其交付给 Reducer 的工作,后者则决定应该将具体的输出交付给哪一个 Reducer。

有多个 Reducer 存在时,MR 框架需要将每个 Mapper 的每项输出,即所收集到的 KV 对,

按某种条件分拣送往不同的 Reducer。这样就把每个 Mapper 的输出划分成了多个分区 (Partition), 有几个 Reducer, 就把每个 Mapper 的输出分成几个 Partition, 而 Partitioner 就起着分拣的作用。分拣所依据的条件不同, 具体的 Partitioner 就不同。比方说, HashPartitioner 就是对每个输出 KV 对中的键值进行简单的 Hash 计算, 根据 Hash 值将其分发给不同的 Reducer。

所以, 在创建 NewOutputCollector 时的构造函数中, 就要把具体的 collector 和 partitioner 一起创建好。

Hadoop 的代码中定义了一个界面 MapOutputCollector<K, V>, 凡是实现了这个界面的类, 除 init() 和 close() 之外, 还必须提供 collect() 和 flush() 这两个函数, 具体的 collector 必须实现这个界面。这里对 createSortingCollector() 的调用, 就是要创建具体的 collector。之所以说 SortingCollector, 是因为它对收集到的 KV 对加以排序。我们看一下这是怎么创建的:

```
[YarnChild.main() > PrivilegedExceptionAction.run() > MapTask.run() > runNewMapper()
> NewOutputCollector() > MapTask.createSortingCollector()]

createSortingCollector(JobConf job, TaskReporter reporter)
> context = new MapOutputCollector.Context(this, job, reporter)
> collectorClasses = job.getClasses(JobContext.MAP_OUTPUT_COLLECTOR_CLASS_ATTR,
    MapOutputBuffer.class) //如果未加设置,就默认为 MapOutputBuffer.class
> for (Class clazz : collectorClasses) { //逐一试验所设置的 collectorClasses
>+ subclazz = clazz.asSubclass(MapOutputCollector.class) //必须实现 MapOutputCollector 界面
>+ LOG.debug("Trying map output collector class: " + subclazz.getName())
>+ collector = ReflectionUtils.newInstance(subclazz, job) //创建 collector 对象
>+ collector.init(context) //初始化,实际上是 MapTask.MapOutputBuffer 的初始化
>+ return collector //如果过程中并未发生异常,那就成功了。
> }
```

具体采用什么 collector, 是可以在配置文件 mapred-default.xml 中加以设置的, 这里的 MAP_OUTPUT_COLLECTOR_CLASS_ATTR 即“mapreduce.job.map.output.collector.class”。如果文件中未加设置就默认为 MapOutputBuffer。事实上, 在 Hadoop 代码中自带的这个配置文件中, 在“mapreduce.job.map.output.collector.class”这个属性下所列的也正是“org.apache.hadoop.mapred.MapTask\$MapOutputBuffer”, 所以实际创建的 collector 就是 MapTask.MapOutputBuffer。这个类定义于 MapTask 内部, 并实现了 MapOutputCollector 界面。

但是可想而知, 如果我们另写一个实现了 MapOutputCollector 界面的 Collector, 并修改配置文件 mapred-default.xml 中对此配置项的设置, 那么这里创建的就可以不是 MapTask.MapOutputBuffer。那样, createSortingCollector() 所创建的其实也可以是一个不带排序的 Collector。我们知道, MR 框架之所以不是数据流架构而是 workflow 架构, 就是因为 Mapper 与 Reducer 之间的排序, 因为 Sort 只有在全部数据都到位之后才能完成, 这就好像在一条河流上筑起了拦河坝。但是 MR 框架其实也为我们留下了在这方面做出改变的余地, 那

就是另写一个不排序的 Collector, 用来替代 MapOutputBuffer, 不过我们现在并不关心这个问题, 还是接着往下看 runNewMapper()。

创建了 collector 和 partitioner 之后, 就是 Context 了。从前面代码摘要中可以看到, MapTask 在调用 mapper.run() 时作为参数传递的是 mapperContext, 这个对象的类型是 WrappedMapper.Context, 其来历是这样的: 代码中首先创建了一个 MapContextImpl 类的对象 mapContext; 又创建了一个 WrappedMapper 类对象, 姑且称之为 wrappedmapper, 这是对 Mapper 的扩充, 顾名思义就是对 Mapper 的包装, 区别就是在其内部定义了一个 Context 类, 把一个经过扩充的 Mapper.Context 包装在 Mapper 内部, 这就是 WrappedMapper.Context。然后, 以此 mapContext 为参数调用 WrappedMapper.getMapContext(), 那样就创建了这个 WrappedMapper.Context 类对象。这个 WrappedMapper.Context 类的摘要如下:

```
class WrappedMapper.Context extends Mapper<KEYIN, VALUEIN,
KEYOUT, VALUEOUT>.Context {}
] MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> mapContext
//MapContext 是个界面, MapContextImpl 类实现了这个界面
] Context(MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> mapContext)
> this.mapContext = mapContext
] getInputSplit()
> return mapContext.getInputSplit() == MapContext.getInputSplit()
] getCurrentKey()
> return mapContext.getCurrentKey() == MapContext.getCurrentKey()
>> return reader.getCurrentKey() == RecordReader.getCurrentKey() //来自 RecordReader
] getCurrentValue()
> return mapContext.getCurrentValue() == MapContext.getCurrentValue()
>> return reader.getCurrentValue() == RecordReader.getCurrentValue() //来自 RecordReader
] nextKeyValue()
> return mapContext.nextKeyValue() == MapContext.nextKeyValue()
>> return reader.nextKeyValue() == RecordReader.nextKeyValue() //也来自 RecordReader
] write(KEYOUT key, VALUEOUT value)
> mapContext.write(key, value) == MapContext.write(key, value)
] ...
```

WrappedMapper.Context 是对 Mapper.Context 的扩充。其内部成分 mapContext, 即其构造函数 Context() 中的 this.mapContext, 就设置成这个 MapContextImpl 类对象 mapContext。那么, WrappedMapper.Context 对 Mapper.Context 的扩充又是些什么呢? 那都是一些操作方法, 例如 write()、getCurrentKey()、nextKeyValue() 等。

于是, 传给 mapper.run() 的 context 就是个 WrappedMapper.Context 对象, 而其中的成分 mapContext 则是个 MapContextImpl 对象。

知道了这些, 我们再来细看 Mapper.map() 中的 context.write():

```
[MapTask.run() > runNewMapper() > Mapper.run() > Mapper.map()]
```

```

Mapper.map(KEYIN key, VALUEIN value, Context context)
> context.write((KEYOUT) key, (VALUEOUT) value) == WrappedMapper.Context.write()
>> mapContext.write(key, value) == MapContextImpl.write(key, value)
    == TaskInputOutputContextImpl.write(key, value)
>>> output.write(key, value) == NewOutputCollector.write(key, value)
>>>> partition = partitioner.getPartition(key, value, partitions) // 确定属于哪一个 Partition
    // 不同的 partitioner 有不同的算法, 例如 HashPartitioner 就采用 Hash 算法
>>>> collector.collect(key, value, partition)
    == MapTask.MapOutputBuffer.collect(K key, V value, int partition)

```

我们知道这里的 context 是个 WrappedMapper.Context 对象, 所以 context.write() 其实就是 WrappedMapper.Context.write(), 这个函数转而调用其内部成分 mapContext 的 write() 函数, 而 mapContext 是个 MapContextImpl 对象, 所以实际调用的是 MapContextImpl.write()。然而 MapContextImpl 类的定义中并未提供 write() 函数, 但是 MapContextImpl 乃是对 TaskInputOutputContextImpl 的扩充, 所以从前者继承了它的 write() 方法。也就是说, mapContext.write() 其实就是 TaskInputOutputContextImpl.write()。然后, 这个 write() 函数又转而调用其内部的 output.write(), 这 output 是个 RecordWriter 对象, 实际上就是前面所说的 NewOutputCollector。而 NewOutputCollector.write(), 则转而调用 collector.collect()。我们在前面看到, 这个 collector 就是 MapTask.MapOutputBuffer。

注意, 从 TaskInputOutputContextImpl.write() 转入 MapTask.MapOutputBuffer.collect() 这一步上增加了一个调用参数 partition, 这是一个指明 KV 对去向的整数, 其作用相当于数组下标, MR 框架中有几个 Reducer, 就有几个 partition。对于 map() 输出的每一个具体的 KV 对, 由一个预先设定、在前面 runNewMapper() 中创建的 partitioner 确定其属于哪一个 partition。而所用的 partitioner, 则是在提交作业时通过 job.setPartitionerClass() 加以设定; 如果未加设定就默认采用 HashPartitioner, 即根据 K 的 Hash 值和 partition 的总数确定把这个 KV 对放在哪一个 partition 中。除对 K 值的 Hash 之外, 也可以根据 K 值或 V 值所处的区间或其他特征确定 KV 对的去向, Hadoop 的代码中提供了好几种 partitioner, 都是对抽象类 Partitioner 的扩充。当然, 用户也可以根据自己写一个 Partitioner, 例如在示例 TeraSort.java 中就自定义了 SimplePartitioner 和 TotalOrderPartitioner 这样两个 Partitioner。

这真是一个曲折的调用过程。之所以要这样曲曲折折地转辗调用, 间接再间接, 为的就是灵活性, 但是运行效率当然要受一些影响, 程序的可读性更受影响。

但是不管怎样, 下面就是 MapTask.MapOutputBuffer 的事了。如前所述, 它实现了 MapOutputCollector 界面。

10.4 作为 Collector 的 MapOutputBuffer

如前所述, Mapper 的输出是通过其 RecordWriter 写出去的, 这个 RecordWriter 从框架或数据流下游看来就是用来收集 Mapper 输出的 Collector, 而 MapOutputBuffer 就是一种对所流经的 KV 对实施排序的 Collector。MapOutputBuffer 类定义于 MapTask 内部。

先看一下 MapTask.MapOutputBuffer 宏观的摘要:

```
class MapTask.MapOutputBuffer<K extends Object, V extends Object>
    implements MapOutputCollector<K, V>, IndexedSortable {}

] int partitions                //输出数据 partition 的总数,即 Reducer 的总数
] TaskReporter reporter         //用于向上级报告任务进展
] CombinerRunner<K,V> combinerRunner    //Combine 环节的 Runner 对象
] CombineOutputCollector<K, V> combineCollector    //Combiner 环节的 collector
] byte[] kvbuffer;              // main output buffer,显然这是个 KV 对缓冲区
] byte[] b0 = new byte[0]       //用于串行化时的临时缓冲,类似于草稿纸
] IndexedSorter sorter          //用于排序的 Sorter 对象
] BlockingBuffer bb = new BlockingBuffer()
    //实际上是个输出流,提供将缓冲区用于 K/V 值的串行化和写入的操作方法
] SpillThread spillThread = new SpillThread() //专门将缓冲区内容写入 Spill 文件的线程
] FileSystem rfs                //存放 Spill 文件的文件系统
] init(MapOutputCollector.Context context) //MapOutputBuffer 的初始化
] collect(K key, V value, final int partition) //将一个 KV 对写入这个 MapOutputBuffer
] flush()                       //写入最后一个 KV 对以后,关闭缓冲区时所需的冲刷
] class BlockingBuffer extends DataOutputStream {} // BlockingBuffer 的类型定义
] BlockingBuffer() //BlockingBuffer 的构造函数
    > super(new Buffer()) //其核心是 Buffer 和下述的 kvbuffer
] class Buffer extends OutputStream {} //提供 BlockingBuffer 的 write()和 flush()等操作
] write(byte b[], int off, int len) //will block if the spill thread is running and it cannot write
] flush()
```

这里的 init()、collect()、flush()和 close()都是界面 MapOutputCollector 规定要提供的函数。结构成分 bb 是个 BlockingBuffer 类对象,这个类定义于 MapOutputBuffer 内部,意为“阻塞式的缓冲区”,其实是个 DataOutputStream,是对 DataOutputStream 的扩充。而 BlockingBuffer,则又是以定义于 MapOutputBuffer 内部的另一个类 Buffer 为基础的。

MapOutputBuffer 的核心就在于其内部定义的这个 Buffer 类和一个充当着缓冲区存储介质的字节数组 kvbuffer。前者提供对此缓冲区进行操作的方法,后者则是物理意义上的缓冲区。

还有个重要的成分是 Spill 线程 SpillThread。这个线程专门负责在缓冲区被充满之后将其内容“溅出(Spill)”到文件系统上的“Spill 文件”中去。一个 Mapper 的输出可能会形成好多 Spill 文件。

前面在通过 createSortingCollector()创建 collector 的时候,我们看到有对 collector.init()的调用,这就是 MapTask.MapOutputBuffer.init()。通过这段代码的摘要我们可以进一步了解这个输出缓冲区的大致构成。

```
[MapTask.run() > runNewMapper() > NewOutputCollector() > createSortingCollector()
> MapOutputBuffer.init()]
```

```

MapOutputBuffer.init(MapOutputCollector.Context context)
> TaskReporter reporter = context.getReporter()    //用来向上级报告进度
> mapTask = context.getMapTask()
> mapOutputFile = mapTask.getMapOutputFile()
    //必须是对抽象类 MapOutputFile 的某种扩充,默认为 MROutputFiles
> spilledRecordsCounter = reporter.getCounter(TaskCounter.SPILLED_RECORDS)
> partitions = job.getNumReduceTasks()    //有几个 Reducer 就分为几个 Partition
> rfs = ((LocalFileSystem)FileSystem.getLocal(job)).getRaw() //Spill 文件所在目录
> spillper = job.getFloat(JobContext.MAP_SORT_SPILL_PERCENT, (float)0.8)
    //Spill 门槛值,百分比,默认为 80%
> sortmb = job.getInt(JobContext.IO_SORT_MB, 100)
    //用于排序的缓冲区大小,默认为 100MB
> sorter = ReflectionUtils.newInstance(job.getClass("map.sort.class",
    QuickSort.class, IndexedSorter.class), job)
    //用于排序的算法,默认为 QuickSort
> maxMemUsage = sortmb << 20
    //左移 20 位正好是 1M。如果 sortmb 是 100,maxMemUsage 就是 100M
> maxMemUsage -= maxMemUsage % METASIZE    // METASIZE 是 16。
    //必须是 16 的整数倍(但既然左移了 20 位,就当然是对齐的,多此一举?)
> kvbuffer = new byte[maxMemUsage] //创建物理缓冲区,缓冲区大小默认为 100M
> bufvoid = kvbuffer.length    //这个缓冲区的上界
> kvmeta = ByteBuffer.wrap(kvbuffer).order(ByteOrder.nativeOrder()).asIntBuffer()
    //先将数组 kvbuffer 包装成一个字节缓冲区 ByteBuffer
    //再设置好在此缓冲区中存储整数时的字节顺序(BigEndian 或 LittleEndian)
    //并为其建立一个用作整数缓冲区 IntBuffer 的视图(View),以便整数的读写
> setEquator(0)    //将分隔点设置在 0 位上,见后面正文的讲解
> bufstart = bufend = bufindex = equator    //此时的 ByteBuffer 视图为空
> kvstart = kvend = kvindex    //IntBuffer 视图也为空
> maxRec = kvmeta.capacity() / NMETA
    //最大记录数是 IntBuffer 的容量除以 16,因为每个 NMETA 的大小是 16
> serializationFactory = new SerializationFactory(job)    //用于 KV 对的串行化
> keySerializer = serializationFactory.getSerializer(keyClass) //用于其中 K 值的串行化
> keySerializer.open(bb) //见前,bb 是个 BlockingBuffer,将其用于 K 值串行化后的输出
> valSerializer = serializationFactory.getSerializer(valClass) //用于其中 V 值的串行化
> valSerializer.open(bb) //也将缓冲区 bb 用于 V 值串行化
> if (job.getCompressMapOutput()) { //如果 Mapper 的输出需要压缩
>+ codecClass = job.getMapOutputCompressorClass(DefaultCodec.class)
>+ codec = ReflectionUtils.newInstance(codecClass, job) //创建编解码器对象
> }
> combinerRunner = CombinerRunner.create(job, getTaskID(),

```

```

        combineInputCounter, reporter, null) //创建 CombinerRunner
>> cls = (Class<?extends Reducer<K,V,K,V>>) job.getCombinerClass() //老 API 中的设置
>> if (cls!= null) { //如果设置了用于老 API 的“mapred.combiner.class”属性
>>+ return new OldCombinerRunner(cls, job, inputCounter, reporter)
>> }
>> taskContext = new org.apache.hadoop.mapreduce.task.TaskAttemptContextImpl(job,
                                                                    taskId, reporter)

>> newcls = taskContext.getCombinerClass() //新 API 中的设置
>> if (newcls!= null) { //如果设置了用于新 API 的“mapreduce.job.combine.class”属性
>>+ return new NewCombinerRunner<K,V>(newcls, job,
                                        taskId, taskContext, inputCounter, reporter, committer)
                                        //返回 NewCombinerRunner,newcls 则是具体的 Combiner
>> }

>> return null // 如果未加设置,则 combinerRunner 有可能是 null
> if (combinerRunner!= null) { //若使用 CombinerRunner 就须创建其 collector
>+ combineCollector = new CombineOutputCollector<K,V>(combineOutputCounter,
                                                        reporter, job)
                                // CombineOutputCollector 实现 OutputCollector 界面,提供 collect()函数
> } else {
>+ combineCollector = null //不用 combinerRunner,就无须 combineCollector
> }

> minSpillsForCombine = job.getInt(JobContext.MAP_COMBINE_MIN_SPILLS, 3)
                                //如果 Spill 文件的数量未达到门槛值(默认为 3),就无须 Combine
> spillThread.setDaemon(true) //将 Spill 线程设置成 Daemon,即断开其 stdin/stdout
> spillThread.start() //启动 Spill 线程
>> spillThread.run() //Spill 线程的 run()函数
>>> while (true) {
>>>+ spillDone.signal()
>>>+ while (!spillInProgress) spillReady.await() //等待需要 Spill 的通知
>>>+ sortAndSpill() //通过 MapTask.sortAndSpill()将缓冲区的内容排序后写入 Spill 文件
>>> }
> while (!spillThreadRunning) spillDone.await() //等待,确认 Spill 线程已在运行才返回

```

MapOutputBuffer 的核心,其物理意义上的“缓冲区”,就是这里所创建的字节数组 kvbuffer。Mapper 每次调用 map()所输出的 KV 对首先第一站就存放在这里。不过存放时不仅要存放 K 和 V 本身,还须存放关于 K 和 V 的“元数据(Meta Data)”,包括 K 值在缓冲区中的起点、V 值的起点、V 值的长度以及 KV 对所属的 partition。其中 K 和 V 的值都是按字节存放的,元数据却是作为整数存放的。为了便于整数操作,这里为 kvbuffer 另外创建了一个作为整数缓冲区(IntBuffer)的视图(View)。而对于这个物理缓冲区的种种操作,则定义于作为内部成分的 Buffer 类中。

Mapper 输出的 K 和 V 都是内存中的对象,实际上是数据结构,需要加以“串行化

(Serialize)”才能进入线性的缓冲区,所以这里要分别创建 `keySerializer` 和 `valSerializer`。两个 `Serializer` 都要先 `open()`,与具体的缓冲区 `bb` 挂上钩,从而也就与 `kvbuffer` 挂上了钩。

另外,在 `Mapper` 一侧还可能要有 `Combine` 环节,如果配置文件中配置,或者具体的应用程序中有设置,例如通过 `Job.setCombinerClass()` 加以设置,那就要创建具体的 `Combiner` 和 `CombinerRunner`,见代码摘要中的 `CombinerRunner.create()`。不过,即使指定了 `Combiner`,如果所形成的 `Spill` 文件不多,这个 `Combine` 环节就可能被跳过。

至于 `Spill` 线程 `spillThread`,则一经创建并启动就在其 `run()` 函数中进入循环,一有通知到来就从 `spillReady.await()` 中被唤醒(此时 `spillInProgress` 已被设置成 `true`),进行一次 `sortAndSpill()`,把缓冲区的内容排序并 `Spill` 到 `Spill` 文件中,以腾出缓冲区空间。这样,每个 `Spill` 文件中的诸多 `KV` 对就是排好序的,最后把这些 `Spill` 文件合并(`Merge`)成一个排好序的大文件,就是一个具体 `MapTask` 在整个 `Map` 阶段的产出。

缓冲区 `kvbuffer` 的默认大小是 100MB,所以每个 `Spill` 文件虽然到不了 100MB,却也离此不远,通常都会有数十兆字节。几个 `Spill` 文件合并在一起,就更大了,这还只是单个 `Mapper` 的输出。

10.5 环形缓冲区 kvbuffer

如上所述, `MapOutputBuffer` 的核心是 `kvbuffer`。这是个字节数组,即 `byte[]`,被用作一个特殊的环形缓冲区,这里先介绍其工作原理。

一般而言,对于缓冲区的最简单的使用方法是:逐次将输出数据写入缓冲区,到缓冲区中剩余的空间已不足以容纳本次输出时,就将整个缓冲区的内容 `Spill` 到文件中,腾出缓冲区的空间,再继续往里面写。但是,在将缓冲区内容 `Spill` 到文件中去的过程中,对于缓冲区的写入就被“阻塞”了。对于默认大小为 100MB 的缓冲区,把这么大的一块数据写入磁盘所需的时间还是挺可观的,于是 `Mapper` 就只好工作一段时间、停顿一段时间,这样的流显然是很不均匀、很不流畅的。为了避免采用这样的全同步方式,可以另外有个 `Spill` 线程,让 `Mapper` 线程源源不断地往缓冲区中写,而让 `Spill` 线程把缓冲区中的内容写入文件。这样, `Mapper` 的输出就变成不阻塞的异步方式。但是,不言而喻,不能让 `Mapper` 对缓冲区的写入与 `Spill` 线程从缓冲区的读出互相干扰。所以,一般都把缓冲区做成环形缓冲区,让写入者在前面跑,读出者在后面追;如果写入者在前面跑得太快,一圈下来追上了读出者,就让写入者停一下;反之如果读出者追得太快,追上了写入者的尾巴,就让读出者停一下。另一种办法就是所谓“双缓冲”,就是设两个缓冲区,写入者写满一个缓冲区就把它交给读出者,让其从中读出,自己则开始写另一个缓冲区。到写入者把一个缓冲区写满的时候,读出者也把另一个缓冲区腾出来了,于是就交换一下。当然,如果二者的速度不匹配,先完成操作的一方就得等一下。这样,就像“跷跷板”一样,二者都交替对两个缓冲区之一进行操作,“井水不犯河水”。从空间使用的角度看,环形缓冲区的方法优于双缓冲,因为双缓冲固定划分两个缓冲区,不能互相调剂。

环形缓冲区只适用于写入和读出的内容保持顺序的条件下,要不然就不能均匀地向前推进。然而在 `Hadoop` 中由 `Mapper` 输出的数据在写入 `Spill` 文件之前是要经过排序的,后面我们会看到,这个排序并不意味着必须要对缓冲区中的这些 `KV` 对在缓冲区中腾来挪去,而可以只改变把它们写入 `Spill` 文件时的次序。可见,在这样的条件下采用环形缓冲区其实并非理想

的方案,因为完全有可能最先写入缓冲区的 KV 对经过排序恰恰要到最后才写出,从而拦住了写入者向前推进的路。那么双缓冲呢?那更不理想。我们在前面看到,用于物理缓冲区的空间大小默认为 100MB,如果分成两块则每块 50MB。那就有可能会这样:来了一个 KV 对,与当前缓冲区中的内容加在一起就超过了 50MB,所以一时写不进去;但是其实另一个缓冲区中可能还有足够的空间(但还没有完成全部写出),就是因为不能把两个缓冲区的空间合在一起使用,就只好让 Mapper 阻塞等待了。所以双缓冲的方案不够灵活,缺乏弹性,更何况同样也有因排序而来的问题。

另外,KV 对经串行化并写入缓冲区之后,就是一串二进制的字节,KV 对的长度又是可变的,说不定上一个 KV 对只有几十字节,而下一个 KV 对却有几兆字节(比方说是整本书的内容),那么怎样确定这些 KV 对的边界和位置呢?怎样访问、怎样排序呢?读者也许马上会想起指针、索引一类的东西:我们可以搞一个结构数组,为写入缓冲区中的每个 KV 对都准备一个数据结构,里面有 K 值起点、V 值起点、V 值长度(K 值的长度就是两个起点之差)、所属 partition 等信息。这样,如果我们要访问缓冲区中的第 N 个 KV 对,就可以从这个数组中的第 N 个(下标为 N-1)数据结构获取其位置信息。这样的数据结构及其所形成的数组,里面是“关于数据的数据”,称为“元数据(Meta Data)”。事实上 Hadoop 在 kvbuffer 中正是采用了类似于这样的方法,每项这样的元数据中有 4 个 32 位整数,共 16 字节,这 4 个整数的内容为:

```
int VALSTART = 0;      // val offset in acct,第一个整数是 V 值起点字节的下标
int KEYSTART = 1;      // key offset in acct,第二个整数是 K 值起点字节的下标
int PARTITION = 2;     // partition offset in acct,第三个整数是 KV 对所属的 Partition
int VALLEN = 3;        // length of value,第四个整数是 V 值的长度
```

这 4 个常数都是相应字段在一项元数据中的位移量,比方说 PARTITION 的位移为 2,就表示在该项元数据在作为整数数组的缓冲区中的起点下标上加 2,就是其 Partition 字段所在处的下标。这样,一个 KV 对在缓冲区中的位置一旦确定,作为 KV 对位置和属性描述的 16 字节的元数据项不管跑到哪里都总是指向这个 KV 对。元数据就是对 KV 对位置和属性的描述。

显然,元数据本身也很重要。那么怎样存储元数据呢?也放在同一个缓冲区中?还是另外搞一个用作结构数组的小缓冲区?如果是后者,那么这个结构数组应该有多大?须知 KV 对的大小可以变化很大:如果 K 和 V 的类型都是整数,那么一个 KV 对只是 8 个字节,而其元数据(按上述)倒有 16 字节,这意味着 100MB 的数据缓冲区得配上 200MB 的元数据缓冲区才行。但是如果 K 为整数而 V 为 Text,那么一个 KV 对说不定就有几兆甚至几十兆字节,从而数据缓冲区中或许只能容纳两三个 KV 对,元数据缓冲区只需很小就行。可是这都是事先不知道、无法预测的。这样想来,最好还是把元数据与数据存放在同一个缓冲区中,让它“水涨船高”,以得到最大的灵活性和空间利用率。

Hadoop 的设计者显然考虑到了这些问题,他们的方案是采用环形缓冲区,并且把元数据 and 数据都放在这个环形缓冲区中。当然,这需要有合理而巧妙的设计。下面我们先看一下基本的思路。

首先,如前所见,在 MapOutputBuffer.init()中通过“new byte[maxMemUsage]”分配空间

设立了一个作为字节数组的缓冲区 kvbuffer, 这个缓冲区是物理存在的。然后, 因为每项元数据都是(32 位)整数, 又通过“`ByteBuffer.wrap(kvbuffer).asIntBuffer()`”为 kvbuffer 建立了一个虚拟的整数缓冲区, 即作为 `IntBuffer` 类的视图(View), 以便元数据的读写。这里面还有对存放次序的考虑, 因为一个整数的四个字节在存放次序上有 Big Endian 和 Little Endian 之分, 如果与硬件所提供的次序不同就会影响效率。

那么怎样存放数据(KV 对)和元数据呢? 这里要引入一个称为“分隔点(equator)”的概念。正是这个分隔点, 在缓冲区中把数据和元数据分隔开来, 如图 10-1 所示。

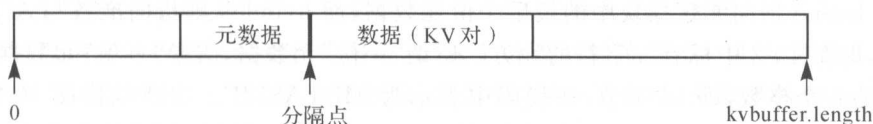


图 10-1 分割点的概念

图 10-1 中的长条矩形表示作为字节数组的缓冲区 kvbuffer, 其起点处的下标为 0, 终点处的下标为 kvbuffer.length。注意, 这是按环形缓冲区使用的, 所以往里写入内容时一旦超过终点就又“翻折”到缓冲区的起点(如果已经空闲), 反之亦然(如果反向伸展的话)。

中间加粗的竖线代表分隔点, 分隔点的位置可以在缓冲区的任何位置上。分隔点的位置确定之后, 凡是数据(KV 对)都放在它的右侧, 即其上方, 并向右伸展; 而元数据则放在它的左侧, 即其下方, 并向左(反向)伸展。写入缓冲区的每个 KV 对都有一组配套的元数据指明其位置和长度。KV 对的长度是可变的, 但是元数据的长度是固定的, 都是 16 个字节, 即 4 个整数。这样, 所有的元数据合在一起就是一个元数据块, 相当于一个(倒立的)数组, 原则上凭下标就可找到某个 KV 对的元数据, 再按照其元数据的指引就可找到这个 KV 对的 K 和 V, 还可以知道这个 KV 对属于哪一个 Partition。

初始化的时候, 分隔点设置在缓冲区的起点, 即下标为 0 的地方。由于是环形缓冲区, 到缓冲区中已经有了一些 KV 对的时候, 就成为如图 10-2 所示的图形(暂时忽略 bufindex 等新加的元素)。

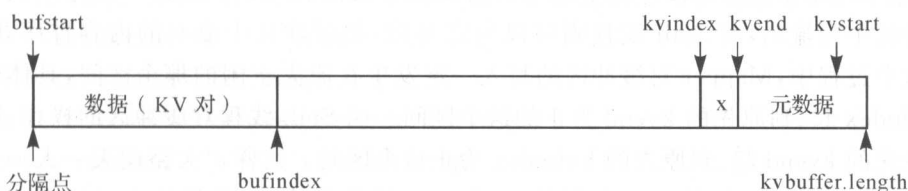


图 10-2 初始化时的分割点设置

就是说, 这时候的数据在缓冲区底部, 自底向上伸展; 元数据则在顶部, 自顶向下伸展; 二者相向而行, 剩余的空闲部分在中间。

不过, 取决于分隔点的位置, 有时候元数据会被分成两块, 一部分在缓冲区的底部, 一部分被翻折到缓冲区的顶端。如果我们把缓冲区看成一个环, 下标为 0 处是圈成环时的“接缝”所在, 那么这时候的元数据块只是跨接缝而已, 没有什么不好理解。可想而知, 此时的分隔点一定比较靠近底部了。同样的道理, 也有可能是数据块跨接缝, 那时的分隔点应该比较靠近顶

部。但是当然,二者不可能同时都跨接缝。此外,缓冲区的空闲部分也有可能因为跨接缝而被分成两段。

虽然图 10-1 和图 10-2 所表示的缓冲区状况似乎很不相同,但是对于环形缓冲区而言逻辑上其实是一回事。

现在,假定这个缓冲区中已经有了一些数据和元数据,那么变量 `bufstart` 指向数据块的起点,变量 `bufindex` 则指向现有数据块的末尾,有新的数据到来时就从这个位置开始继续往里写。对于元数据,则有变量 `kvstart`、`kvend` 和 `kvindex`。其中,`kvstart` 指向元数据块中的第一份元数据,`kvend` 指向元数据块中的最后一份元数据,而 `kvindex` 则指向准备写入下一份元数据的位置,即图 10-2 中标有 `x` 字符的地方。所谓“一份”元数据,或者“一项”元数据,是关于一个 KV 对的 4 个整数,即 16 字节,在代码中表示为 `METASIZE`。之所以像图 10-2 所示的那样,是因为虽然元数据是从上向下伸展,但 CPU 对内存的读写操作却总是从某个地址开始向上伸展。所以,如果又有一个 KV 对到来,就得为之准备一份元数据,将其写入 `kvindex` 所指的地方,然后将 `kvindex` 减 4,使其指向准备写入再下一份元数据的位置;`kvend` 也要相应下移一个位置,但仍保持跟 `kvindex` 相差一份元数据的距离。而 KV 对数据本身,则写入 `bufindex` 所指的位置,然后使 `bufindex` 加上这个 KV 对的长度,指向准备写入再下一个 KV 对的位置。注意,在写入 KV 对的时候我们把缓冲区视作字节数组,`bufindex` 是字节数组内的下标;而在写入元数据时则把缓冲区视作整数数组,`kvstart`、`kvend` 和 `kvindex` 都是整数数组内的下标。

其实 `kvstart`、`kvindex` 这些变量的名字取得不好,因为不管是元数据还是数据本身都是有关 KV 对的,`kvstart` 并未特别指明这是用于元数据还是 KV 对本身,还不如叫 `metastart` 更为清晰。

随着数据的到来,数据块和元数据块相向伸展,缓冲区中的空闲部分就越越来越小,最后终将缩小到不足以容纳下一个 KV 对其元数据的地步,这时候显然就得把缓冲区的内容写回到 Spill 文件中了。但是,到了这个时候才开始 Spill 就太迟了,因为此时 Mapper 只好停下来等待,直到 Spill 完成才能恢复,这又像是同步的操作了。

所以 Spill 操作应该在缓冲区中的空间尚未耗尽之前就开始。那样,Mapper 线程还可继续往缓冲区中写输出,而 Spill 线程则可以与之并发,把缓冲区中原有的内容排序并 Spill 出去。在这个过程中,Mapper 对缓冲区的写入一定发生在原先空闲的那个区间,具体就是从原先的 `bufindex` 起、到原先的 `kvend` 为止的这个区间。而 Spill 线程对缓冲区的操作,则一定发生在从原先的 `kvend` 起、到原先的 `bufindex` 为止这个区间。这样,“大路朝天一人一边”,就可以做到“井水不犯河水”。那么这个所谓“空间尚未耗尽之前”究竟是什么时候呢?回到前面 `MapOutputBuffer.init()` 的代码摘要中,那里有个变量 `spillper`,就代表着启动 Spill 的门槛值。这个变量的值是可以配置的,配置文件 `mapred-default.xml` 中的相应属性名为“`mapreduce.map.sort.spill.percent`”,若未加设定就默认为 0.8,即 80%。也就是说,按默认的门槛值,当缓冲区达到 80% 满的时候就应开始排序和 Spill。排序意味着次序和位置的调整,但是只要调整元数据项的位置就可以了,数据块的内容无须变动。

Spill 完成之后,原先的 `kvend` 到 `bufindex` 之间的这个区间就释放了,而原先空闲的区间则可能已经有了一些内容(数据和元数据)。但请注意,原先的分隔点 `equator` 现在已经失去了意义,其所在的位置现在已属空闲区间;而新写入的数据和元数据,则不符合二者“背靠背”

相连以 equator 分隔的格局,而是二者“背对背”但中间隔开了一段距离,更确切地说就是隔开了整个因 Spill 而腾出来的空闲区间。显然,此时的数据或者元数据,这二者之一必须挪个地方,搬迁到可以使二者背靠背相连的地方,而中间就是新的分隔点。搬迁谁呢?当然是搬元数据比较简单。

但是这里又有个问题,就是新的数据块起点,也即 Spill 之前的 bufindex 所指的位置,未必就是与整数边界对齐的;而新的元数据块的起点,则必须与整数边界对齐。于是,经过一次 Spill 之后,数据块与元数据块也许不再能“无缝”地背靠背相连了,中间可能会有个小小的空隙。当然,这个空隙一定可以小于一个整数,即 4 个字节的大小。这样,一般而言,经过一次 Spill 之后,可以用新的数据块起点作为分隔点,这个分隔点的位置不一定与整数边界对齐(不一定是 4 的倍数);其右侧(上方)就是数据块,但是其左侧(下方)的元数据块的起点却必须跟整数边界对齐,因而可能与分隔点之间有个小小的空隙。

明白了这些原理和思路,我们就可以看具体的代码摘要了。

10.6 对 MapOutputBuffer 的输出

如前所述,具体 Mapper 在其 map() 方法中所调用的 context.write(), 最终转化落实到了 MapOutputBuffer 的 collect() 方法上。

```
[Mapper.map() > context.write() => collector.collect() == MapOutputBuffer.collect()]
```

```
MapOutputBuffer.collect(K key, V value, final int partition)
```

```
> reporter.progress()
```

```
> bufferRemaining -= METASIZE //METASIZE = NMETA * 4, NMETA = 4;
```

```
//bufferRemaining 的计算已包含 80% 临界的考虑
```

```
> if (bufferRemaining <= 0) { //剩余缓冲区空间变小已抵临界,需要 Spill
```

```
>+ do { //一次 Spill
```

```
>++ if (!spillInProgress) { //如果已经在做 Spill, 当然就不需要做什么了
```

```
>+++ int kvbidx = 4 * kvindex
```

```
>+++ int kvbend = 4 * kvend
```

```
>+++ int bUsed = distanceTo(kvbidx, bufindex)
```

```
>+++> return i <= j?j-i:mod-i+j
```

```
>+++ boolean bufsoftlimit = bUsed >= softLimit
```

```
>+++ if ((kvbend + METASIZE) % kvbuffer.length != equator - (equator % METASIZE)) {
```

```
//spill finished, reclaim space, 现在 Spill 已经完成, 需要释放空间
```

```
//这个条件说明缓冲区中原来是有数据的(元数据块非空)
```

```
>+++++ resetSpill()
```

```
>+++++> int e = equator
```

```
>+++++> bufstart = bufend = e //表示 KV 对数据块为空
```

```
>+++++> int aligned = e - (e % METASIZE)
```

```
>+++++> kvstart = kvend = (int) (((long)aligned - METASIZE + kvbuffer.length)
```

```

% kvbuffer.length) / 4
//表示元数据块为空
>+++++ bufferRemaining = Math.min(distanceTo(bufindex, kvbidx)
- 2 * METASIZE, softLimit - bUsed) - METASIZE
//重新计算 bufferRemaining, softLimit 为 0.8 * kvbuffer.length
>+++++ continue
>++++ } else if (bufsoftlimit && kvindex != kvend){
>+++++ startSpill() //启动 Spill
>+++++> kvend = (kvindex + NMETA) % kvmeta.capacity()
>+++++> bufend = bufmark
>+++++> spillInProgress = true
>+++++> LOG.info("Spilling map output")
>+++++> spillReady.signal() //通知 SpillThread, 开始 Spill
>+++++ int avgRec =
(int)(mapOutputByteCounter.getCounter()/mapOutputRecordCounter.getCounter())
//计算这些 KV 对的平均长度
>+++++ int distkvi = distanceTo(bufindex, kvbidx)
>+++++ int newPos = (bufindex + Math.max(2 * METASIZE - 1,
Math.min(distkvi / 2, distkvi / (METASIZE + avgRec) * METASIZE))) % kvbuffer.length
>+++++ setEquator(newPos)
>+++++> equator = pos
>+++++> int aligned = pos - (pos % METASIZE)
>+++++> kvindex = (int) (((long)aligned - METASIZE + kvbuffer.length) % kvbuffer.length) / 4
>+++++> LOG.info("(EQUATOR) " + pos + " kvi " + kvindex + "(" + (kvindex * 4) + ")")
>+++++ bufmark = bufindex = newPos
>+++++ int serBound = 4 * kvend
>+++++ bufferRemaining = Math.min(distanceTo(bufend, newPos),
Math.min(distanceTo(newPos, serBound), softLimit)) - 2 * METASIZE
//重新计算 bufferRemaining, softLimit 为 0.8 * kvbuffer.length
>++++ } //end else if
>++ } //end if (!spillInProgress)
>+ } while (false) //这个 do{}while() 只做一遍
> } //end if (bufferRemaining <= 0)
> //serialize key bytes into buffer, 空间一般并未真正用尽, 还可以往里写(但不能保证)
> // serialize key bytes into buffer
> int keystart = bufindex
> keySerializer.serialize(key) //先把 Key 串行化并写入 bb
> if (bufindex < keystart) { // wrapped the key; must make contiguous
>+ bb.shiftBufferedKey() == BlockingBuffer.shiftBufferedKey()
>+> //Set position from last mark to end of writable buffer,

```

```

    //then rewrite the data between last mark and kvindex.
>+> int headbytelen = bufvoid - bufmark
>+> bufvoid = bufmark
>+> int kvbidx = 4 * kvindex
>+> int kvbend = 4 * kvend
>+> int avail = Math.min(distanceTo(0, kvbidx), distanceTo(0, kvbend))
>+> if (bufindex + headbytelen < avail) {
>+>+ System.arraycopy(kvbuffer, 0, kvbuffer, headbytelen, bufindex)
    //arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
>+>+ System.arraycopy(kvbuffer, bufvoid, kvbuffer, 0, headbytelen)
>+>+ bufindex += headbytelen
>+>+ bufferRemaining -= kvbuffer.length - bufvoid
>+> } else {
>+>+ byte[] keytmp = new byte[bufindex]
>+>+ System.arraycopy(kvbuffer, 0, keytmp, 0, bufindex)
>+>+ bufindex = 0
>+>+ out.write(kvbuffer, bufmark, headbytelen) == ?.write(kvbuffer, bufmark, headbytelen)
>+>+ out.write(keytmp)
>+> }
>+ keystart = 0
> }
> // serialize value bytes into buffer
> int valstart = bufindex
> valSerializer.serialize(value) //再把 Value 串行化并写入 bb
> bb.write(b0, 0, 0) == BlockingBuffer.write(b0, 0, 0) //注意写出的数据长度为 0
>> DataOutputStream.write(byte[] b, int off, int len)
    //见原注释:It's possible for records to have zero length, ...,这里只是为检验边界条件
> int valend = bb.markRecord() == BlockingBuffer.markRecord()
    //as the metadata for this record are not yet written
>> bufmark = bufindex
>> return bufindex
> mapOutputRecordCounter.increment(1) //修改统计信息,增加了一个记录
> mapOutputByteCounter.increment(distanceTo(keystart, valend, bufvoid)) //增加的字节数
    //填写元数据:
> kvmeta.put(kvindex + PARTITION, partition)
> kvmeta.put(kvindex + KEYSTART, keystart)
> kvmeta.put(kvindex + VALSTART, valstart)
> kvmeta.put(kvindex + VALLEN, distanceTo(valstart, valend))
> kvindex = (kvindex - NMETA + kvmeta.capacity()) % kvmeta.capacity()
    //调整 kvindex

```

```

> catch (MapBufferTooSmallException e) { //发生因缓冲区太小所致的异常
    //< MapTask.Buffer.write(byte b[], int off, int len) < collect()
>+ LOG.info("Record too large for in-memory buffer: " + e.getMessage())
>+ spillSingleRecord(key, value, partition) //KV 对太大,缓冲区太小,直接 Spill 出去
>+ mapOutputRecordCounter.increment(1)
> }

```

前面已经讲过这循环缓冲区的原理,这里又加了注释,就不再详细讲述了,读者可以结合源码自己阅读理解。

10.7 Sort 和 Spill

我们在前面的摘要中看过 Spill 线程的 run() 函数,每当这个线程被 spillReady.signal() 唤醒的时候,就会调用 sortAndSpill(), 将缓冲区的内容排序后写入 Spill 文件(另外在对缓冲区进行 flush() 的时候也会调用这个函数)。

```
[MapTask.MapOutputBuffer.spillThread.run() > sortAndSpill()]
```

```
MapTask.MapOutputBuffer.sortAndSpill()
```

```

> size = distanceTo(bufstart, bufend, bufvoid) + partitions * APPROX_HEADER_LENGTH
    // 环形缓冲区中的数据长度,加上每个 Partition 一个 HEADER 的长度
> spillRec = new SpillRecord(partitions)
> Path filename = mapOutputFile.getSpillFileForWrite(numSpills, size)
>> attempt_id = conf.get(JobContext.TASK_ATTEMPT_ID)
>> path = String.format(String.format(SPILL_FILE_PATTERN, attempt_id, spillNumber)
    // 格式为 "%s_spill_%d.out", spillNumber 就是 numSpills
>> lDirAlloc.getLocalPathForWrite(path, size, conf)
> out = rfs.create(filename) // 创建 Spill 文件
> int mstart = kvend / NMETA // num meta ints, 计算共有几项元数据,以最后一项为起点
> int mend = 1 + (kvstart >= kvend? kvstart : kvmeta.capacity() + kvstart) / NMETA
    // 计算第一项元数据的下标,把整个缓冲区看作一个元数据数组
> sorter.sort(MapOutputBuffer.this, mstart, mend, reporter) == IndexedSorter.sort()
    == QuickSort.sort() // 对元数据项排序
>> sortInternal(...) // 我们对具体的排序方法不感兴趣
----- 至此已按 Partition 和 Key 排序 -----
> int spindex = mstart // 从最后一项(缓冲区中属于元数据块的数组下标最小处)开始
> IndexRecord rec = new IndexRecord() // 创建索引记录
> InMemValBytes value = new InMemValBytes()
> for (int i=0; i < partitions; ++i) { // 对于每一个 Partition
>+ long segmentStart = out.getPos() // 获得 Spill 文件上的当前位置
>+ FSDataOutputStream partitionOut = CryptoUtils.wrapIfNecessary(job, out)

```

```

>+> if (isShuffleEncrypted(conf)) { //如果要加密
>+>+ cryptoCodec = CryptoCodec.getInstance(conf)
>+>+ bufferSize = getBufferSize(conf)
>+>+ ...
>+>+ return new CryptoFSDDataInputStream(in, cryptoCodec,
                                         bufferSize, getEncryptionKey(), iv)
                                         //在 out,即 Spill 文件基础上创建加密通道,返回后就成为 partitionOut
>+> } else { //如果不加密
>+>+ return in //如果不加密那就是上面的 out 本身。这里 in 为形参,实参是上面的 out
>+> }
>+ writer = new Writer<K, V>(job, partitionOut, keyClass, valClass,
                             codec, spilledRecordsCounter)
                             //创建以 Spill 文件为目标的加密或不加密 Writer
>+ if (combinerRunner == null) { // spill directly,不经过 combinerRunner
>++ key = new DataInputBuffer()
>++ while (spindex < mEND &&
           kvmeta.get(offsetFor(spindex % maxRec) + PARTITION) == i) {
           //扫描全部元数据,如果某项元数据中的 PARTITION 字段表明该记录属于 Partition_i
>+++ int kvoff = offsetFor(spindex % maxRec) //maxRec = kvmeta.capacity() / NMETA
>+++ keystart = kvmeta.get(kvoff + KEYSTART) //key 所在的位置
>+++ valstart = kvmeta.get(kvoff + VALSTART) //val 所在的位置
>+++ key.reset(kvbuffer, keystart, valstart - keystart) == DataInputBuffer.reset()
           //将串行化了的 K 值转移到 DataInputBuffer 中
>+++> this.buf = input //kvbuffer
>+++> this.count = start + length //keystart + (valstart - keystart) = valstart
>+++> this.mark = start //keystart
>+++> this.pos = start
>+++ getVBytesForOffset(kvoff, value) //see above: value = new InMemValBytes()
           //将串行化了的 K 值转移到 InMemValBytes 中
>+++> int vallen = kvmeta.get(kvoff + VALLEN)
>+++> vbytes.reset(kvbuffer, kvmeta.get(kvoff + VALSTART), vallen)
>+++ writer.append(key, value) //写入 Spill 文件
>+++ ++ spindex //往前推进
>++ } //end while
>+ } else { //如果要经过 combinerRunner
>++ int spstart = spindex
>++ while (spindex < mEND && kvmeta.get(offsetFor(spindex % maxRec) + PARTITION) == i)
           ++ spindex //直到碰上属于下一个 partition 的数据
>++ if (spstart != spindex) { //如果有数据
           //would like to avoid the combiner if we've fewer than

```



```

//some threshold of records for a partition.
>+++ combineCollector.setWriter(writer) == CombineOutputCollector.setWriter(writer)
>+++> this.writer = writer //使 writer 成为 CombineOutputCollector 的 writer
>+++ kvIter = new MRResultIterator(spstart, spindex)
>+++ combinerRunner.combine(kvIter, combineCollector) == NewCombinerRunner.combine()
//在写出过程中加上 combine() 环节, combineCollector 为 CombineOutputCollector
>+++> reducer = ReflectionUtils.newInstance(reducerClass, job)
>+++> converter = new OutputConverter(collector)
>+++> reducerContext = createReduceContext(reducer, job, taskId, iterator, null, inputCounter,
converter, committer, reporter, comparator, keyClass, valueClass)
>+++> reducer.run(reducerContext) == Reducer.run()
>++ }
>+ } //end else, 经过 combinerRunner
>+ writer.close() //关闭 Writer, 当前这个 Partition 已完成 Spill
>+ rec.startOffset = segmentStart //记录在 IndexRecord 中, 一个 Partition 一个索引记录
>+ rec.rawLength = writer.getRawLength() + CryptoUtils.cryptoPadding(job)
>+ rec.partLength = writer.getCompressedLength() + CryptoUtils.cryptoPadding(job)
>+ spillRec.putIndex(rec, i) //将 partition i 的 IndexRecord 放入 SpillRecord
> } //end for each partition, 对这个 Partition 的处理结束
- - - - - 排序和 Spill 已经完成, 还需处理一下索引块 - - - - -
> if (totalIndexCacheMemory >= indexCacheMemoryLimit) { //create spill index file
//索引块太大, 需要创建并写入 Spill 索引文件
>+ Path indexFilename = mapOutputFile.getSpillIndexFileForWrite(
numSpills, partitions * MAP_OUTPUT_INDEX_RECORD_LENGTH)
>+ spillRec.writeToFile(indexFilename, job) //将 SpillRecord 写入 indexFile
> } else { //索引块不大, 可以留在内存中
>+ indexCacheList.add(spillRec)
>+ totalIndexCacheMemory += spillRec.size() * MAP_OUTPUT_INDEX_RECORD_LENGTH
> } //end for (int i = 0; i < partitions; ++ i)
> LOG.info("Finished spill " + numSpills)
>++ numSpills //增加了一次 Spill

```

正如函数名所示, 这是先 sort, 再 Spill。排序, 是对起着指针索引作用的元数据排序, KV 对数据本身则保持在原地不动。排序以后元数据块还是元数据块, 但是里面各项元数据的位置排好了序。如前所述, 元数据是对 KV 对位置和属性的描述, 只要保证不被拆散, 就与所在位置无关。

这里的排序, 是先按 Partition, 然后在同一个 Partition 内按 Key 值排序。所以排完序以后 Partition 相同的元数据都挨在一起, 第 0 号 Partition 的在最前面, 再按 Key 值大小排序。所以排完序后就可以用 for 循环内嵌 while 循环, 对元数据块一遍扫描下来, 就可逐个 Partition、逐个 KV 对地把所有 KV 对都写出到 Spill 文件中, 若有需要还可加密。此外, 如果有 Combine, 还可以在逐项写出 KV 对的途中插入 Combine 环节, Combine 都是用

Reducer 实现的。最后,在 Spill 的过程中还会产生一个索引块,这个索引块缓存在内存中,如果其大小达到了 `indexCacheMemoryLimit`,就把它写入索引文件。结合上述的原理和我添加在摘要中的注释,读者对这个流程应该不难理解。

我们并不关心具体的排序算法,具体的算法可以配置,默认为 QuickSort。

然而,看着程序中的语句“`sorter.sort(MapOutputBuffer.this, mstart, mend, reporter)`”,尽管我们知道这个 `sorter` 是 QuickSort,但怎么就能断定它是按 Partition 和 Key 值排序的呢?

其实,QuickSort 一类标准排序算法的实现都是通用的,具体按什么内容排取决于两个回调函数,即 `compare()` 和 `swap()`。这里调用 `sort()` 时的第一个参数是 `MapOutputBuffer.this`,就表示回调函数是 `MapOutputBuffer.compare()` 和 `MapOutputBuffer.swap()`。我们不妨看一下这两个函数。首先是 `compare()`:

```
[MapTask.MapOutputBuffer.spillThread.run() > sortAndSpill() > sort() =>
MapOutputBuffer.compare()]
```

```
public int compare(final int mi, final int mj) {
    final int kvi = offsetFor(mi % maxRec);
    final int kvj = offsetFor(mj % maxRec);
    final int kvip = kvmeta.get(kvi + PARTITION); //先比较 PARTITION 字段的内容
    final int kvjp = kvmeta.get(kvj + PARTITION);
    // sort by partition
    if (kvip != kvjp) {
        return kvip - kvjp; //如果两个 Partition 不同,那就已经比出了先后
    }
    // sort by key,如果 Partition 相同就再比两个 Key 的内容
    return comparator.compare(kvbuffer,
        kvmeta.get(kvi + KEYSTART), //KEYSTART 字段指向 Key 的起点
        kvmeta.get(kvi + VALSTART) - kvmeta.get(kvi + KEYSTART), //Key 的长度
        kvbuffer, //缓冲区的起点
        kvmeta.get(kvj + KEYSTART),
        kvmeta.get(kvj + VALSTART) - kvmeta.get(kvj + KEYSTART));
}
```

显然这是先按 PARTITION 字段的内容比,如果二者相同再按元数据所指向的 Key 值(给定起点和长度)比。比较之后,如果需要交换两项元数据的位置,就调用 `swap()`:

```
[MapTask.MapOutputBuffer.spillThread.run() > sortAndSpill() > sort() =>
MapOutputBuffer.swap()]
```

```
public void swap(final int mi, final int mj) {
    int iOff = (mi % maxRec) * METASIZE; //元数据 1 的起点
    int jOff = (mj % maxRec) * METASIZE; //元数据 2 的起点
    System.arraycopy(kvbuffer, iOff, META_BUFFER_TMP, 0, METASIZE);
```

```

System.arraycopy(kvbuffer, jOff, kvbuffer, iOff, METASIZE);
System.arraycopy(META_BUFFER_TMP, 0, kvbuffer, jOff, METASIZE);
}

```

可见交换是按一项元数据的大小 METASIZE 进行的,是整项元数据的交换,不会把一项元数据拆散。

就这样,只要 Mapper 还在运行,有数据输出,collect()和 sortAndSpill()就会并存,直至 Mapper 处理完所有的输入数据。

10.8 Map 计算的终结与 Spill 文件的合并

当 Mapper 的数据源中不再有输入数据时,mapper.run()中的 while 循环会因为调用 context.nextKeyValue()返回 false 而结束,于是便返回到 runNewMapper()。在那里,程序会关闭输入通道和输出通道。关闭输出通道意味着关闭 Collector,关闭前先要“冲刷”一下。

```
[MapTask.runNewMapper() > NewOutputCollector.close() > MapOutputBuffer.flush()]
```

```

MapOutputBuffer.flush()
> LOG.info("Starting flush of map output")
> while (spillInProgress) { //若有 Spill 正在进行,则等待其结束
>+ reporter.progress()
>+ spillDone.await()
> }
> checkSpillException()
> kvbend = 4 * kvend //kvend 是元数据块的终点,元数据块是向下伸展的
                        //kvend 是以整数计的数组下标,kvbend 是以字节计的数组下标
> if ((kvbend + METASIZE) % kvbuffer.length != equator - (equator % METASIZE)) {
                        //说明缓冲区中原来是有数据的,现在 Spill 已经完成,需要释放空间
                        //spill finished,一次 Spill 刚完成,需要调整一些参数以释放缓冲区空间
>+ resetSpill()
>+ int e = equator
>+ bufstart = bufend = e
>+ int aligned = e - (e % METASIZE)
>+ kvstart = kvend =
                        (int)((long)aligned - METASIZE + kvbuffer.length) % kvbuffer.length / 4
>+ LOG.info("(RESET) equator " + e + " kv " + kvstart + "(" + (kvstart * 4) + ")"
                        + " kvi " + kvindex + "(" + (kvindex * 4) + ")")
> }
> if (kvindex != kvend) { //缓冲区非空,需要再 Spill 一次
>+ kvend = (kvindex + NMETA) % kvmeta.capacity()
>+ bufend = bufmark

```

```

>+ LOG.info("Spilling map output")
>+ sortAndSpill()
> }
----- 至此,缓冲区已空,所有数据都已在 Spill 文件中 -----
> spillThread.interrupt()    //使 Spill 线程不再运行
> spillThread.join()        //结束 Spill 线程
> mergeParts()              //合并 Spill 文件
> Path outputPath = mapOutputFile.getOutputFile()
> fileOutputByteCounter.increment(rfs.getFileStatus(outputPath).getLen())

```

冲刷的目的,首先是要让缓冲区中所有的 KV 对数据都进入 Spill 文件,缓冲区中不应再有数据。然后,因为每次 Spill 都会产生 Spill 文件,Spill 文件很可能不止一个,所以要把所有 Spill 文件的内容合并到单个文件中,以备分发(Shuffle)给各个 Reducer。

所以,如果有 Spill 正在进行,当然应该等待其完成;但是也可能没有 Spill 正在进行。总而言之,只要缓冲区非空,就得再来一次 sortAndSpill()。

所有数据都在 Spill 文件中以后,就可以通过 mergeParts()合并 Spill 文件了。

```

[MapTask.runNewMapper() > NewOutputCollector.close() > MapOutputBuffer.flush() >
MapOutputBuffer.mergeParts()]

```

```

MapOutputBuffer.mergeParts()
> filename = new Path[numSpills]    //每次 Spill 都有个文件,所以数组大小为 numSpills
> for(int i=0; i < numSpills; i++) {    //统计所有这些文件合并之后的大小
>+ filename[i] = mapOutputFile.getSpillFile(i)    //获取文件名
>+ finalOutFileSize += rfs.getFileStatus(filename[i]).getLen()    //获取文件大小
> }
> if (numSpills == 1) { //the spill is the final output    //如果只有一次 Spill,那就简单了
>+ sameVolRename(filename[0], mapOutputFile.getOutputFileForWriteInVolume(filename[0]))
    //换个文件名,在原名上加后缀 file.out
>+ if (indexCacheList.size() == 0) {    //索引块缓存 indexCacheList 已空
>++ sameVolRename(mapOutputFile.getSpillIndexFile(0),
    mapOutputFile.getOutputIndexFileForWriteInVolume(filename[0]))
    //SpillIndexFile 改名
>+ } else { //索引块缓存 indexCacheList 中还有索引记录,要写到索引文件
>++ indexCacheList.get(0).writeToFile(
    mapOutputFile.getOutputIndexFileForWriteInVolume(filename[0]), job)
>+ }
>+ sortPhase.complete()
>+ return    //如果只有一个 Spill,这就已完成“合并”
> }
-----

```

```

> for (int i = indexCacheList.size(); i < numSpills; ++i) { //Spill 文件不止一个,需要合并
>+ Path indexFileName = mapOutputFile.getSpillIndexFile(i)
>+ indexCacheList.add(new SpillRecord(indexFileName, job))
//先将所有 SpillIndexFile 收集在一起
> } //end for
> finalOutFileSize += partitions * APPROX_HEADER_LENGTH
//每个 partition 都有个 Header
> finalIndexFileSize = partitions * MAP_OUTPUT_INDEX_RECORD_LENGTH
//IndexFile, 每个 partition 一个记录
> Path finalOutputFile = mapOutputFile.getOutputFileForWrite(finalOutFileSize)
> Path finalIndexFile = mapOutputFile.getOutputIndexFileForWrite(finalIndexFileSize)
> FSDataOutputStream finalOut = rfs.create(finalOutputFile, true, 4096)
//创建(合并后的)最终输出文件
-----
> if (numSpills == 0) { //要是压根就没有生成 SpillFile,也要创建空文件
>+ IndexRecord rec = new IndexRecord() //创建索引记录
>+ SpillRecord sr = new SpillRecord(partitions) //创建 Spill 记录
>+ for (int i = 0; i < partitions; i++) {
>++ long segmentStart = finalOut.getPos()
>++ FSDataOutputStream finalPartitionOut = CryptoUtils.wrapIfNecessary(job, finalOut)
>++ Writer<K, V> writer = new Writer<K, V>(job, finalPartitionOut, keyClass,
//valClass, codec, null)
>++ writer.close() == IFile.Writer.close() //创建后马上关闭,形成空文件
>++ rec.startOffset = segmentStart
>++ rec.rawLength = writer.getRawLength() + CryptoUtils.cryptoPadding(job)
>++ rec.partLength = writer.getCompressedLength() + CryptoUtils.cryptoPadding(job)
>++ sr.putIndex(rec, i)
>+ }
>+ sr.writeToFile(finalIndexFile, job) //将索引记录写入索引文件
>+ finalOut.close()
>+ sortPhase.complete()
>+ return
> } //end if (numSpills == 0)
-----
> sortPhase.addPhases(partitions); // Divide sort phase into sub-phases
> IndexRecord rec = new IndexRecord()
> SpillRecord spillRec = new SpillRecord(partitions)
> for (int parts = 0; parts < partitions; parts++) { //对于每一个 Partition
>+ segmentList = new ArrayList<Segment<K, V>>(numSpills)
//create the segments to be merged

```

```

>+ for(int i = 0; i < numSpills; i++) { //准备合并其所有 Spill 文件
>+ IndexRecord indexRecord = indexCacheList.get(i).getIndex(parts)
>+ Segment<K,V> s = new Segment<K,V>(job, rfs, filename[i], indexRecord.startOffset, ...)
>+ segmentList.add(i, s) //把每个 Spill 文件中属于同一 Partition 的区段位置收集起来
>+ } //end for
>+ int mergeFactor = job.getInt(JobContext.IO_SORT_FACTOR, 100)
//The number of streams to merge at once while sorting
>+ boolean sortSegments = segmentList.size() > mergeFactor
//sort the segments only if there are intermediate merges
>+ RawKeyValueIterator kvIter = Merger.merge(job, rfs, keyClass, valClass, codec,
segmentList, mergeFactor, new Path(mapId.toString()),
job.getOutputKeyComparator(), reporter, sortSegments, null,
spilledRecordsCounter, sortPhase.phase(), TaskType.MAP)
//合并同一 Partition 在所有 Spill 文件中的内容,也可能还需要 sort
//合并的结果是一个序列 kvIter
>+ // write merged output to disk
>+ segmentStart = finalOut.getPos()
>+ FSDataOutputStream finalPartitionOut = CryptoUtils.wrapIfNecessary(job, finalOut)
>+ Writer<K, V> writer = new Writer<K, V>(job, finalPartitionOut,
keyClass, valClass, codec, ...)
>+ if (combinerRunner == null || numSpills < minSpillsForCombine) { //如果无须 combine,
>+ Merger.writeFile(kvIter, writer, reporter, job) //就将合并的结果直接写入文件:
>+> while(records.next()) { //records == kvIter
>+> writer.append(records.getKey(), records.getValue())
>+> }
>+ } else { //如果需要 combine
>+ combineCollector.setWriter(writer) //那就插入 combine 环节
>+ combinerRunner.combine(kvIter, combineCollector) //合并结果经 combine 后写入文件
>+ }
>+ writer.close() //然后关闭 Write 通道
>+ sortPhase.startNextPhase()
>+ rec.startOffset = segmentStart //从当前段的起点开始
>+ rec.rawLength = writer.getRawLength() + CryptoUtils.cryptoPadding(job) //长度
>+ rec.partLength = writer.getCompressedLength() + CryptoUtils.cryptoPadding(job)
>+ spillRec.putIndex(rec, parts)
> } //end for(int parts = 0; parts < partitions; parts++)
-----
> spillRec.writeToFile(finalIndexFile, job) //将 spillRec 写入合并的 IndexFile
> finalOut.close() //关闭最终输出文件
> for(int i = 0; i < numSpills; i++) {

```



```
>+ rfs.delete(filename[i],true)           //删掉所有的 Spill 文件
> }
```

Mapper 完成对所有输入文件的处理,并将缓冲区中的数据写出到 Spill 文件之后,Spill 文件的存在只有三种可能:没有 Spill 文件;只有一个 Spill 文件;有多个 Spill 文件。这里三种情况都考虑到了,不管是哪一种情况,都会有一个最终输出文件。这个最终文件的格局跟单个 Spill 文件是一样的,就是按 Partition 分成若干区段,每个 Partition 都有个头部,然后是排好序的 KV 数据。这个 Merge 操作结合原先为各个 Spile 文件进行的 Sort,就构成一次 MergeSort。这个 MergeSort 只是针对同一 Mapper 的多个 Spill 文件。以后在 Reducer 那里还会有 Merge,那就是针对众多 Mapper 的 MergeSort 了。

我们并不关心具体的 Merge 算法和有关细节,摘要中又加了注释,读者对这个函数的代码应该不难理解。

当 MapTask 完成所有的处理,从 runNewMapper() 返回后,下一步操作是 done(),开始 MapTask 的收尾和善后。MapTask 的收尾和善后涉及怎样把 MapTask 所生成的输出数据文件交给 ReduceTask 的问题。注意 MapTask 和 ReduceTask 都是对 Task 的扩充,但是二者都未定义自己的 done() 函数,所以它们调用的都是 Task.done()。我们就从 Task.done() 开始往下看:

```
[MapTask.run() > Task.done()]

Task.done(TaskUmbilicalProtocol umbilical, TaskReporter reporter)
> updateCounters()
> boolean commitRequired = isCommitRequired()
> if (commitRequired) {
>+ retries = MAX_RETRIES
>+ setState(TaskStatus.State.COMMIT_PENDING)
>+ while (true) {
>++ umbilical.commitPending(taskId, taskStatus)
>++ break //如果在 commitPending()期间并未发生异常就 break,否则重试
>+ }
>+ commit(umbilical, reporter, committer) //wait for commit approval and commit
> }
> taskDone.set(true)
> reporter.stopCommunicationThread()
> updateCounters()
> sendLastUpdate(umbilical)
> sendDone(umbilical) //signal the tasktracker that we are done
>> umbilical.done(getTaskID()) == TaskAttemptListenerImpl.done(getTaskID())
//向 MRAppMaster 上的 TaskAttemptImpl 发送 TA_DONE 事件
>>> attemptID = TypeConverter.toYarn(taskAttemptID)
>>> taskHeartbeatHandler.progressing(attemptID)
```

```
>>> context.getEventHandler().handle(
new TaskAttemptEvent(attemptID, TaskAttemptEventType.TA_DONE))
```

在这 TA_DONE 事件的驱动下, 相应 TaskAttemptImpl 对象的状态机先执行 CleanupContainerTransition.transition(), 然后转入 SUCCESS_CONTAINER_CLEANUP 状态。状态机中的跳变是这样的:

```
static final CleanupContainerTransition CLEANUP_CONTAINER_TRANSITION =
new CleanupContainerTransition()
```

```
addTransition(TaskAttemptStateInternal.RUNNING,
TaskAttemptStateInternal.SUCCESS_CONTAINER_CLEANUP,
TaskAttemptEventType.TA_DONE, CLEANUP_CONTAINER_TRANSITION)
```

这个跳变操作向 ContainerLauncher 发出 CONTAINER_REMOTE_CLEANUP 请求:

```
[MapTask.run() > Task.done() > TaskAttemptEventType.TA_DONE
=> TaskAttemptImpl.CleanupContainerTransition.transition()]
```

```
CleanupContainerTransition.transition()
```

```
> taskAttempt.taskAttemptListener.unregister(taskAttempt.attemptId, taskAttempt.jvmID)
> if (event instanceof TaskAttemptKillEvent) {
>+ taskAttempt.addDiagnosticInfo(((TaskAttemptKillEvent) event).getMessage())
> }
> taskAttempt.reportedStatus.progress = 1.0f
> taskAttempt.updateProgressSplits()
> taskAttempt.eventHandler.handle(
new ContainerLauncherEvent(taskAttempt.attemptId,
taskAttempt.container.getId(),
StringInterner.weakIntern(taskAttempt.container.getNodeId().toString()),
taskAttempt.container.getContainerToken(),
ContainerLauncher.EventType.CONTAINER_REMOTE_CLEANUP))
//向 ContainerLauncher 发出 CONTAINER_REMOTE_CLEANUP 请求
```

注意, 这里的 TaskAttemptEventType.TA_DONE 事件是由具体 MapTask 在其所在节点上发出的, 但是所引起的状态机跳变却是在 MRAppMaster 所在的节点上, 这两个节点之间有着“脐带”般的连系。对于 MapTask 而言, umbilical 就代表着 MRAppMaster。

我们以前看过 ContainerLauncher 对于 CONTAINER_REMOTE_LAUNCH 的反应, 这一次则是 CONTAINER_REMOTE_CLEANUP:

```
ContainerLauncherImpl.EventProcessor.run()
> ContainerId containerID = event.getContainerID()
> Container c = getContainer(event)
> switch(event.getType()) {
```

```

> case CONTAINER_REMOTE_LAUNCH:
>+ ContainerRemoteLaunchEvent launchEvent = (ContainerRemoteLaunchEvent) event
>+ c.launch(launchEvent)
>+ break
> case CONTAINER_REMOTE_CLEANUP:
>+ c.kill() == ContainerLauncherImpl.kill() //杀掉当前 Task,在我们这个情景中是 MapTask
>+> if(this.state == ContainerState.PREP) {
>+>+ this.state = ContainerState.KILLED_BEFORE_LAUNCH
>+> } else if (!isCompletelyDone()) {
>+>+ ContainerManagementProtocolProxyDataproxy =
        getCMPProxy(this.containerMgrAddress, this.containerID)
        //获取通向相应 ContainerManagerImpl 对象的 proxy
>+>+ List<ContainerId> ids = new ArrayList<ContainerId>()
>+>+ ids.add(this.containerID)
>+>+ StopContainersRequest request = StopContainersRequest.newInstance(ids)
>+>+ response = proxy.getContainerManagementProtocol().stopContainers(request)
        //对 MapTask 所在 NM 节点上 ContainerManagerImpl 对象的 RPC 调用
        //要求其 stopContainers(),并等待其回应
>+>+ cmProxy.maybeCloseProxy(proxy) //也许不再需要用到这个 proxy
>+>+ this.state = ContainerState.DONE
>+> } //end else if
>+ context.getEventHandler().handle(new TaskAttemptEvent(this.taskAttemptID,
        TaskAttemptEventType.TA_CONTAINER_CLEANED))
//Container 已被 kill(),向这个 TaskAttemptImpl 发出 TA_CONTAINER_CLEANED 事件
>+ break
> }
> removeContainerIfDone(containerID)

```

接到 CONTAINER_REMOTE_CLEANUP 事件, ContainerLauncher 通过 RPC 机制调用 MapTask 所在节点上的 ContainerManagerImpl.stopContainers(), 使这个 MapTask 所属的容器转入 KILLED_BY_APPMASTER 状态而不再活跃。操作成功返回后即向相应的 TaskAttemptImpl 发出 TA_CONTAINER_CLEANED 事件。

这个 Task 所属 TaskAttemptImpl 的状态机对此事件的反应为:

```

addTransition(TaskAttemptStateInternal.SUCCESS_CONTAINER_CLEANUP,
    TaskAttemptStateInternal.SUCCEEDED,
    TaskAttemptEventType.TA_CONTAINER_CLEANED,
    new SucceededTransition())

```

先执行 SucceededTransition.transition(), 然后进入 SUCCEEDED 状态。

```

TaskAttemptImpl.SucceededTransition.transition()
> taskAttempt.setFinishTime()

```

```

> taskAttempt.eventHandler.handle(createJobCounterUpdateEventTASucceeded(taskAttempt))
//有关统计
> taskAttempt.logAttemptFinishedEvent(TaskAttemptStateInternal.SUCCEEDED)
//有关日志
> e1 = new TaskTAttemptEvent(taskAttempt.attemptId, TaskEventType.T_ATTEMPT_SUCCEEDED)
> taskAttempt.eventHandler.handle(e1) //让 TaskAttemptImpl 的状态进一步变化
> e2 = new SpeculatorEvent(taskAttempt.reportedStatus, taskAttempt.clock.getTime())
>> super(Speculator.EventType.ATTEMPT_STATUS_UPDATE, timestamp)
> taskAttempt.eventHandler.handle(e2) //向 DefaultSpeculator 发送状态更新

```

这一步所做的是有关统计和日志的事,同时也向安排作为后备的 DefaultSpeculator 对象发出状态更新。对于 TaskAttemptImpl 本身则发出 T_ATTEMPT_SUCCEEDED 事件,让其状态机进入 SUCCEEDED 状态:

```

addTransition(TaskStateInternal.RUNNING, TaskStateInternal.SUCCEEDED,
    TaskEventType.T_ATTEMPT_SUCCEEDED, new AttemptSucceededTransition())

```

一次 TaskAttempt 成功了,就意味着所尝试的任务也成功了,所以 TaskAttempt 的这个状态变化当然应该波及相应的 TaskImpl 对象。

TaskImpl.AttemptSucceededTransition.transition()

```

> TaskTAttemptEvent taskTAttemptEvent = (TaskTAttemptEvent) event
> TaskAttemptId taskAttemptId = taskTAttemptEvent.getTaskAttemptId()
> task.handleTaskAttemptCompletion(taskAttemptId,
    TaskAttemptCompletionEventStatus.SUCCEEDED)
//向 TaskImpl 对象自身发送 SUCCEEDED 事件,让其进一步执行某些扫尾工作
> task.finishedAttempts.add(taskAttemptId)
> task.inProgressAttempts.remove(taskAttemptId)
> task.successfulAttempt = taskAttemptId
> task.sendTaskSucceededEvents()
>> e = new JobTaskEvent(taskId, TaskState.SUCCEEDED)
>> eventHandler.handle(e) //向相应 JobImpl 对象发送 TaskState.SUCCEEDED 事件
>> LOG.info("Task succeeded with attempt " + successfulAttempt)
>> if (historyTaskStartGenerated) {
>>+ TaskFinishedEvent tfe = createTaskFinishedEvent(this, TaskStateInternal.SUCCEEDED)
>>+ eventHandler.handle(new JobHistoryEvent(taskId.getJobId(), tfe)) //有关历史记录
>> }
> for (TaskAttempt attempt : task.attempts.values()) {
//告诉这个 task 的其他 attempts,现在有个 TaskAttempt 已经率先成功
>+ if (attempt.getID() != task.successfulAttempt && !attempt.isFinished()) {
>+ e = new TaskAttemptKillEvent(attempt.getID(),
    SPECULATION + task.successfulAttempt + " succeeded first!")
>+> super(attemptID, TaskAttemptEventType.TA_KILL)

```

```

> ++ task.eventHandler.handle(e) //向不再需要存在的 TaskAttemptImpl 发送 TA_KILL
> + }
> } //end for
> task.finished(TaskStateInternal.SUCCEEDED) //有关统计

```

这主要是 TaskImpl 的扫尾和善后,包括向其上层,即其所属的 JobImpl 对象发送 TaskState.SUCCEEDED 事件。我们在这里要特别关心一下向 TaskImpl 自身发送 SUCCEEDED 事件所导致的操作,即 TaskImpl.handleTaskAttemptCompletion():

```

[AttemptSucceededTransition.transition() => TaskImpl.handleTaskAttemptCompletion()]

handleTaskAttemptCompletion(TaskAttemptId attemptId,
                             TaskAttemptCompletionEventStatus status)
{
    > TaskAttempt attempt = attempts.get(attemptId)
    > if (attempt.getNodeHttpAddress() != null) {
    > + TaskAttemptCompletionEvent tce =
        recordFactory.newRecordInstance(TaskAttemptCompletionEvent.class)
    > + String scheme = (encryptedShuffle)? "https://" : "http://" //采用 https 或 http
    > + tce.setMapOutputServerAddress(StringInterneter.weakIntern(
        scheme + attempt.getNodeHttpAddress().split(":")[0] + ":" + attempt.getShufflePort())
        //形如 "http://NodeHttpAddress:ShufflePort"
    > + tce.setStatus(status)
    > + tce.setAttemptId(attempt.getID())
    > + if (attempt.getFinishTime() != 0 && attempt.getLaunchTime() != 0)
        runTime = (int)(attempt.getFinishTime() - attempt.getLaunchTime())
    > + tce.setAttemptRunTime(runTime)
    > + e = new JobTaskAttemptCompletedEvent(tce)
    > +> super(completionEvent.getAttemptId().getTaskId().getJobId(),
        JobEventType.JOB_TASK_ATTEMPT_COMPLETED)
    > + eventHandler.handle(e) //将此事件发送给所属的 JobImpl 对象
}

```

这里的 TaskAttemptCompletionEvent 是个界面,实际创建的 tce 是实现了这个界面的 TaskAttemptCompletionEventPBIImpl 类对象,这显然是想要跨节点发送出去的。

我们关心的是对其 setMapOutputServerAddress() 的调用,把本节点 (MapTask 所在节点) 的 MapOutputServer 地址设置成一个 Web 地址。这意味着,MapTask 所留下的数据输出,即合并以后的 Spill 文件,可以用这个地址通过 Http 连接获取。

然后,以此事件 tce 为参数创建一个 JobTaskAttemptCompletedEvent 事件,并将其发送给当前这个 TaskImpl 所属的 JobImpl。此后的活动,这里就不往下追了,有兴趣或需要的读者可以自己阅读源代码。一般而言,一个作业不会只有一个任务,所以仅仅个别任务的完成不会影响整个 Job 的状态(Running)。到了所有任务都完成,或者有任务运行失败而且不可恢复的时候,这个作业的状态才会发生变化。

10.9 Reduce 阶段

由于 Sort 的存在, MapTask 与 ReduceTask 之间是一种工作流的架构, 而不是数据流的架构。在 MapTask 尚未结束运行, 其输出结果尚未排序及合并之前, ReduceTask 是没有数据输入的, 即使此时 ReduceTask 进程已经创建, 也只能睡眠等待有 MapTask 完成运行, 从而可从其所在节点获取其输出数据。如前所述, 一个 MapTask 最终的数据输出是一个合并好的 Spill 文件, 可以通过该节点的 Web 地址, 即所谓的 MapOutputServerAddress 加以访问。

所以, ReduceTask 可以在 MapTask 运行到差不多快完成的时候才启动。不过启动早了也不要紧, 那也只是白白地在所在节点上提早占用了几个容器 (VCore 和内存) 而已, 因为它们实际上不会有什么活动。

ReduceTask 是个线程, 这个线程运行在为 YarnChild 而启动的 Java 虚拟机上。我们就从 ReduceTask.run() 开始看 MapReduce 数据流的 Reduce 阶段。

```
[YarnChild.main() > doAs() > ReduceTask.run()]
```

```
ReduceTask.run(JobConf job, final TaskUmbilicalProtocol umbilical)
> if (isMapOrReduce()) {
>+ copyPhase = getProgress().addPhase("copy")
>+ sortPhase = getProgress().addPhase("sort")
>+ reducePhase = getProgress().addPhase("reduce")
> }
> TaskReporter reporter = startReporter(umbilical)
//start thread that will handle communication with parent
> initialize(job, getJobID(), reporter, useNewApi) == Task.Initialize(job, getJobID(), ...)
> if (jobCleanup) {
>+ runJobCleanupTask(umbilical, reporter)
>+ return //这只是为了 JobCleanup, 做完就行了
> }
> if (jobSetup) {
>+ runJobSetupTask(umbilical, reporter)
>+ return //这只是为了 JobSetup, 做完就行了
> }
> if (taskCleanup) {
>+ runTaskCleanupTask(umbilical, reporter)
>+ return //这只是为了 TaskCleanup, 做完就行了
> }
-----这才是真要成为 Reducer-----
> codec = initCodec()
> ShuffleConsumerPlugin shuffleConsumerPlugin = null
```



```

> combinerClass = conf.getCombinerClass()
> CombineOutputCollector combineCollector = (null != combinerClass)?
    new CombineOutputCollector(reduceCombineOutputCounter, reporter, conf) : null
    //如果需要就创建 combineCollector
> Class<?extends ShuffleConsumerPlugin> clazz =
    job.getClass(MRConfig.SHUFFLE_CONSUMER_PLUGIN, Shuffle.class,
        ShuffleConsumerPlugin.class)
    //在配置文件中寻找“mapreduce.job.reduce.shuffle.consumer.plugin.class”
    //应该是个实现了 ShuffleConsumerPlugin 界面的类,默认 Shuffle.class
> shuffleConsumerPlugin = ReflectionUtils.newInstance(clazz, job)
    //创建 Shuffle 类对象
> LOG.info("Using ShuffleConsumerPlugin: " + shuffleConsumerPlugin)
> ShuffleConsumerPlugin.Context shuffleContext = new ShuffleConsumerPlugin.Context(...)
    //创建其 Context 对象,ShuffleConsumerPlugin.Context
> shuffleConsumerPlugin.init(shuffleContext) == Shuffle.init(shuffleContext)
>> this.localMapFiles = context.getLocalMapFiles()
>>> scheduler = new ShuffleSchedulerImpl<K, V>(jobConf, taskStatus, reduceId,
    this, copyPhase, context.getShuffledMapsCounter(),
    context.getReduceShuffleBytes(), context.getFailedShuffleCounter())
    //创建 Shuffle 所需的调度器
>>> merger = createMergeManager(context) //创建 Shuffle 内部的 merger
>>>> new MergeManagerImpl<K, V>(reduceId, jobConf, context.getLocalFS(),
    ..., context.getMapOutputFile())
    //创建 MergeManagerImpl 对象和 Merger 线程
> rIter = shuffleConsumerPlugin.run() == Shuffle.run()
    //从各个 Mapper 复制其输出文件,并加以合并(排序),等待直至完成
> mapOutputFilesOnDisk.clear()
> sortPhase.complete() //排序阶段结束
> setPhase(TaskStatus.Phase.REDUCE) //进入 REDUCE 阶段
> statusUpdate(umbilical)
> keyClass = job.getMapOutputKeyClass()
> valueClass = job.getMapOutputValueClass()
> RawComparator comparator = job.getOutputValueGroupingComparator()
> if (useNewApi) //新 API
>+ runNewReducer(job, umbilical, reporter, rIter, comparator, keyClass, valueClass)
    //确定、创建,并运行具体的 Reducer
> else //老 API
>+ runOldReducer(job, umbilical, reporter, rIter, comparator, keyClass, valueClass)
> shuffleConsumerPlugin.close() == Shuffle.close() //doing nothing
> done(umbilical, reporter)

```

这里的 `shuffleConsumerPlugin` 是实现了 `ShuffleConsumerPlugin` 界面的某类对象, 具体可以通过配置文件中的“`mapreduce.job.reduce.shuffle.consumer.plugin.class`”选项加以设置, 默认为 `Shuffle`。从代码中可见, 完成 `shuffleConsumerPlugin.run()`, 通常就是 `Shuffle.run()`, 是可以开始 `runNewReducer()` 或 `runOldReducer()` 的先决条件。这是因为, `Shuffle` 类对象就是从 `MapTask` 提取数据的搬运工。而且, `Shuffle` 的搬运方式并非一边搬运一边就让 `Reducer` 处理, 而是要把所有 `MapTask` 的输出数据全都搬到并进行合并排序之后才开始提供给 `Reducer`。

一般而言, `MapTask` 与 `ReduceTask` 是多对多的关系, 我们假定有 M 个 `Mapper` 和 N 个 `Reducer`。每个 `Mapper` 的输出经 `Partitioner` 加以分拣之后分成了 N 份, 分别属于一个具体的分区即 `Partition`, 而每一个 `Reducer` 则承担着一个 `Partition` 的 `Reduce` 操作。这样, 每一个 `Reducer` 都需要从每一个 `Mapper` 节点那里要来属于自己的那一份数据, 这样就是 M 份, 把这 M 份数据合在一起就是一个完整的 `Partition`, 如果需要的话还须排序, 成为这个具体 `Reducer` 的输入数据。这个过程是一个数据搬运和重组的过程, 称为 `Shuffle`。`Shuffle` 这个词有洗牌、发牌的意思, 其实就是数据的搬运和重组, 因为 `Shuffle` 的前后数据所在的地点变了, 组成的方式也变了。`Shuffle` 是个开销颇大的操作, 会给网络造成较大的流量, 因为 N 个 `Partition` 的总和就是整个数据集的全部数据。另外, `Shuffle` 也会带来延迟, 因为 M 个 `Mapper` 的计算有快有慢, 但 `Shuffle` 要到最后一个 `Mapper` 完成后以后才能完成, 而 `Reduce` 操作又要等 `Shuffle` 完成才能开始。当然, 这种延迟也并非 `Shuffle` 本身所造成, 如果 `Reducer` 不需要等整个 `Partition` 的数据全都到位并排序, 那就无须与最慢的 `Mapper` 同步了。这里我们又看到为排序付出的代价。

所以, `Shuffle` 在 `MapReduce` 框架中扮演着很重要的角色。下面是这个类的摘要:

```
class Shuffle<K, V> implements ShuffleConsumerPlugin<K, V>, ExceptionReporter {
    ] ShuffleConsumerPlugin.Context context
    ] TaskAttemptID reduceId
    ] JobConf jobConf
    ] TaskUmbilicalProtocol umbilical
    ] ShuffleSchedulerImpl<K,V> scheduler
    ] MergeManager<K, V> merger
    ] Task reduceTask//Used for status updates
    ] Map<TaskAttemptID, MapOutputFile> localMapFiles
    ] init(ShuffleConsumerPlugin.Context context)
    ] run()
```

上面我们已经在 `ReduceTask.run()` 的摘要中展开了对于 `Shuffle.init()` 的调用, 在那里创建了一个 `ShuffleSchedulerImpl` 对象和一个 `MergeManagerImpl` 对象, 后面我们将看到它们的作用, 但是这二者的名称已经使我们有了一些大致的印象。

此后就是对 `Shuffle.run()` 的调用, 这是很重要的一步。注意, 虽然 `Shuffle` 有一个 `run()` 函数, 它却并非一个线程, 只是用了这么个函数名而已。我们看看这个函数的摘要:

```
[YarnChild.main() > doAs() > ReduceTask.run() > Shuffle.run()]
```

Shuffle.run()

```

> eventFetcher = new EventFetcher<K,V>(reduceId, umbilical,
                                     scheduler, this, maxEventsToFetch)
> eventFetcher.start()           //这个 eventFetcher 是一个线程
> numFetchers = isLocal?
    1 : jobConf.getInt(MRJobConfig.SHUFFLE_PARALLEL_COPIES, 5)
> fetchers = new Fetcher[numFetchers] //创建一个 Fetcher 数组,相当于一个线程池
> if (isLocal) { //如果 Mapper 与 Reducer 在同一机器上,那就只需本地 Fetcher
>+ fetchers[0] = new LocalFetcher<K,V>(jobConf, reduceId, scheduler, ...,
                                     reduceTask.getShuffleSecret(), localMapFiles)
    // LocalFetcher 是对 Fetcher 的扩充,也是线程
>+ fetchers[0].start() //本地 Fetcher 只有一个
> } else { //Mapper 与 Reducer 不在同一机器上,需要若干个跨节点的 Fetcher
>+ for (int i = 0; i < numFetchers; ++ i) { //启动所有的 Fetcher
>++ fetchers[i] = new Fetcher<K,V>(jobConf, reduceId, scheduler, ...,merger, ...,
                                     reduceTask.getShuffleSecret())
    //创建 Fetcher 线程
>++ fetchers[i].start() //跨节点的 Fetcher 可能得要好几个
>+ } //end for
> }
> while (!scheduler.waitUntilDone(PROGRESS_FREQUENCY)) {
    //等待所有 Fetcher 都完成,每次超时就报告一下进度
>+ reporter.progress() //Wait for shuffle to complete successfully
> }
> eventFetcher.shutdown() //Shuffle 操作已完成,所有 MapTask 的输出文件都已拷贝过来
> for (Fetcher<K,V> fetcher : fetchers) fetcher.shutdown() //关闭所有的 Fetcher
> scheduler.close() //不再需要 Shuffle 调度
> copyPhase.complete(); // copy is already complete,文件复制阶段已告结束
-----
> taskStatus.setPhase(TaskStatus.Phase.SORT) //下一步就是 Reducer 一侧的 MergeSort 了
> reduceTask.statusUpdate(umbilical) //通过“脐带”向 MRAppMaster 更新状态
> RawKeyValueIterator kvIter = merger.close() == MergeManagerImpl.close()
    //Finish the on-going merges... 合并和排序,完成后返回一个队列,即 kvIter
> return kvIter

```

要从 MapTask(所在的节点)搬运数据到 ReduceTask,无非就是两种办法,一种是由 MapTask 推送,另一种是由 ReduceTask 提取。这里采用的是后者,由 ReduceTask 主动去 MapTask 那里提取,实际上就是文件复制。

在 Shuffle 进入其 run()函数之前,ReduceTask.run()调用过它的 init()函数,在那里创建了调度器 scheduler 和用于合并排序的 merger。到了 run()函数中,则又会创建一个 EventFetcher 线程和若干个 Fetcher 线程。顾名思义,Fetcher 的作用就是拿取,向 MapTask

所在的节点拿取数据。而 EventFetcher, 虽然也就是 Fetcher, 所拿取的却是 Event, 而不是数据本身。

Fetcher 线程的数量要视情况而定。如果是 Uber 模式, 那么 MapTask 与 ReduceTask 在同一节点上, 并且只有一个 MapTask, 所以 Fetcher 也只需一个, 并且是 LocalFetcher。那是对 Fetcher 的扩充, 是一种特殊的 Fetcher。如果不是 Uber 模式, 那就可能有很多的 MapTask, 并且一般而言与 ReduceTask 不在同一个节点上。在这种情况下 Fetcher 的数量是可以(静态)配置的, 默认为 5 个。数组 fetchers[] 就相当于一个 Fetcher 线程池。

创建了 EventFetcher 线程和 Fetcher 线程池以后, 这个 Shuffle.run() 函数就进入了 while 循环, 但是它在 while 循环中(除报告进度外)什么事也不做, 只是等待。可见实际的操作都是在“体外”由别的那些线程, 即一个 EventFetcher 线程和若干 Fetcher 线程完成的。我们可以猜想, 这里 EventFetcher 线程一定起着某种枢纽的作用, 事实也确实如此。

```
class EventFetcher<K,V> extends Thread {}
] TaskAttemptID reduce
] TaskUmbilicalProtocol umbilical
] ShuffleScheduler<K,V> scheduler
] int maxEventsToFetch
] run()
> while (!stopped && !Thread.currentThread().isInterrupted()) {
>+ int numNewMaps = getMapCompletionEvents() //获取有关 Map 完成的事件
>+> do {
>+>+ MapTaskCompletionEventsUpdate update = umbilical.getMapCompletionEvents(
                (org.apache.hadoop.mapred.JobID)reduce.getJobID(), fromEventIdx,
                maxEventsToFetch, (org.apache.hadoop.mapred.TaskAttemptID)reduce)
                //通过“脐带”从 MRAppMaster 获取有关 Map 完成的事件(报告)
>+>+ events = update.getMapTaskCompletionEvents()
                //从中获取有关具体 MapTask 结束运行的情况
>+>+ fromEventIdx += events.length // Update the last seen event ID
                // Process the TaskCompletionEvents:
                // 1. Save the SUCCEEDED maps in knownOutputs to fetch the outputs.
                // 2. Save the OBSOLETE/FAILED/KILLED maps in obsoleteOutputs to stop
                //    fetching from those maps.
                // 3. Remove TIPFAILED maps from neededOutputs since we don't need their
                //    outputs at all.
>+>+ for (TaskCompletionEvent event : events) { //对于所获取的每个事件报告:
>+>+ scheduler.resolve(event) == ShuffleSchedulerImpl.resolve(event)
>+>+> switch (event.getTaskStatus()) {
>+>+> case SUCCEEDED: //MapTask 成功完成运行
>+>+>+ URI u = getBaseURI(reduceId, event.getTaskTrackerHttp()) //获取其 URI
>+>+>+ addKnownMapOutput(u.getHost() + ":" + u.getPort(), u.toString(),
```

```

        event.getTaskAttemptId())
        //将这个 MapTask 所在的节点主机记录下来,供 Fetcher 线程使用
    >+>+>+> MapHost host = mapLocations.get(hostName)
    >+>+>+> host.addKnownMap(mapId)
    >+>+>+> if (host.getState() == State.PENDING) {
    >+>+>+>+ pendingHosts.add(host)
    >+>+>+>+ notifyAll()
    >+>+>+> }
    >+>+>+> maxMapRuntime = Math.max(maxMapRuntime, event.getTaskRunTime())
    >+>+>+> break
    >+>+> case FAILED: case KILLED: case OBSOLETE: //MapTask 运行失败
    >+>+>+> obsoleteMapOutput(event.getTaskAttemptId())
    >+>+>+> break
    >+>+> case TIPFAILED:
    >+>+>+> tipFailed(event.getTaskAttemptId().getTaskID())
    >+>+>+> }
    >+>+> if (TaskCompletionEvent.Status.SUCCEEDED == event.getTaskStatus()) {
    >+>+>+> ++numNewMaps
    >+>+>+> }
    >+>+> }
    >+> } while (events.length == maxEventsToFetch)
    >+> return numNewMaps
    >+ if (!Thread.currentThread().isInterrupted()) Thread.sleep(SLEEP_TIME)
    > } //end while

```

MapTask 与 ReduceTask 没有直接的联系,MapTask 也不知道 ReduceTask 究竟在哪些节点上,它只是把有关进度的事件报告给 MRAppMaster。而 ReduceTask 则通过“脐带”执行 getMapCompletionEvents()操作向 MRAppMaster 索取有关 MapTask 结束运行的事件报告。尽管其中也可能有失败,但绝大多数的 MapTask 应该能成功完成。凡是成功完成运行的 MapTask,就要有 Fetcher 前去索要其输出数据(在一个合并好的 Spill 文件中)。这个信息就是通过 shcheduler,即 ShuffleSchedulerImpl 转达的。ShuffleSchedulerImpl 并不是线程,只是个普通的对象,普通的模块。

如前所述,fetchers[]就好比线程池,里面有若干线程(默认为 5 个),这些线程就等着来自 EventFetcher 的通知,一旦获知某个 MapTask 已经成功完成运行,就立即前往索要数据。

```

class Fetcher<K,V> extends Thread {
    ] run()
    > while (!stopped && !Thread.currentThread().isInterrupted()) {
    >+ merger.waitForResource() //If merge is on, block
    >+ MapHost host = scheduler.getHost() //Get a host to shuffle from
    //从 scheduler 获取一个已成功完成运行的 MapTask 所在节点

```

```

>+ copyFromHost(host) //Shuffle,从那个节点复制 MapTask 输出数据
>+ scheduler.freeHost(host)
> } //end while (!stopped &&!Thread.currentThread().isInterrupted())

```

总的来说,这些 Fetcher 要做的事就是 Shuffle,不过 Shuffle 也得一个个来,每次处理一个 Map 节点。所以,狭义的 Shuffle()操作就是 MapTask 输出数据的跨节点(或不跨节点)拷贝。MapTask 的输出数据已经排好序按 Partition 分块存储在 Spill 文件中。

```
[Fetcher.run() > copyFromHost()]
```

```

Fetcher.copyFromHost(MapHost host) //这是在 ReduceTask 所在的节点上
> List<TaskAttemptID> maps = scheduler.getMapsForHost(host)
//从 scheduler 获取目标节点上的 MapTask 集合
> Set<TaskAttemptID> remaining = new HashSet<TaskAttemptID>(maps)
//其中有待完成 shuffle 的 MapTask 集合
> URL url = getMapOutputURL(host, maps) //生成针对 MapTask 所在节点的 URL
>> BaseUrl = host.getBaseUrl()
>>> return baseUrl < ShuffleSchedulerImpl.resolve(TaskCompletionEvent event)
>>>> URI u = getBaseURI(reduceId, event.getTaskTrackerHttp())
>>>>> StringBuffer baseUrl = new StringBuffer(url)
>>>>> if (!url.endsWith("/")) baseUrl.append("/")
>>>>> baseUrl.append("mapOutput?job = ")
>>>>> baseUrl.append(reduceId.getJobID()) //添上本作业的 JobID
>>>>> baseUrl.append("&reduce = ")
>>>>> baseUrl.append(reduceId.getTaskID().getId()) //然后是 ReduceTask 的 TaskID
>>>>> baseUrl.append("&map = ") //“map = ”后面暂时空着,由下面的 for()循环填写
>>>>> URI u = URI.create(baseUrl.toString())
>>>>> return u
>>>> addKnownMapOutput(u.getHost() + ":" + u.getPort(), u.toString(),
//在 URL 的“map = ”后面填上 MapTask 的 ID
>> url = new StringBuffer(BaseURL)
>> for (TaskAttemptID mapId : maps) {
>>+ if (!first) url.append(",")
>>+ url.append(mapId)
>> }
>> LOG.debug("MapOutput URL for " + host + " -> " + url.toString())
>> return new URL(url.toString()) //返回生成的 URL
> setupConnectionsWithRetry(host, remaining, url) //与对方建立 HTTP 连接
>> openConnectionWithRetry(host, remaining, url)
>>> openConnection(url)
>>>> HttpURLConnection conn = (HttpURLConnection) url.openConnection()

```



```

>>>> if (sslShuffle) {
>>>>+ HttpURLConnection httpsConn = (HttpURLConnection) conn
>>>>+ httpsConn.setSSLSocketFactory(sslFactory.createSSLSocketFactory())
>>>>+ httpsConn.setHostnameVerifier(sslFactory.getHostnameVerifier())
>>>> }
>>>> connection = conn
>> String msgToEncode = SecureShuffleUtils.buildMsgFrom(url)
>> String encHash = SecureShuffleUtils.hashFromString(msgToEncode, shuffleSecretKey)
>> setupShuffleConnection(encHash)
>>> connection.addRequestProperty(
        SecureShuffleUtils.HTTP_HEADER_URL_HASH, encHash)
>>> connection.setReadTimeout(readTimeout)
>>> connection.addRequestProperty (ShuffleHeader.HTTP_HEADER_NAME,
        ShuffleHeader.DEFAULT_HTTP_HEADER_NAME)
>>> connection.addRequestProperty (ShuffleHeader.HTTP_HEADER_VERSION,
        ShuffleHeader.DEFAULT_HTTP_HEADER_VERSION)
>> connect(connection, connectionTimeout)
>> verifyConnection(url, msgToEncode, encHash)
> input = new DataInputStream(connection.getInputStream()) //将此连接作为输入流
> while (!remaining.isEmpty() && failedTasks == null) {
>+ copyMapOutput(host, input, remaining, fetchRetryEnabled)
>+ header = new ShuffleHeader()
>+ header.readFields(input)
>+ mapId = TaskAttemptID.forName(header.mapId)
>+ compressedLength = header.compressedLength
>+ decompressedLength = header.uncompressedLength
>+ forReduce = header.forReduce
>+ InputStream is = input
>+ is = CryptoUtils.wrapIfNecessary(jobConf, is, compressedLength) //如果需要解压/解密
>+ compressedLength- = CryptoUtils.cryptoPadding(jobConf)
>+ decompressedLength- = CryptoUtils.cryptoPadding(jobConf)
>+ mapOutput = merger.reserve(mapId, decompressedLength, id)
        == MergeManagerImpl.reserve(mapId, decompressedLength, id)
        //为 merger 预留一个 MapOutput: InMemoryMapOutput 或 OnDiskMapOutput
>+>> if (!canShuffleToMemory(requestedSize)) {
        //如果内存中容纳不下就只好放在磁盘上
>+>>+ return new OnDiskMapOutput<K,V>(mapId, reduceId, this, requestedSize, ...)
>+>> }
>+>> return unconditionalReserve(mapId, requestedSize, true) //内存中能容纳
>+>>> usedMemory += requestedSize

```

```

>+>>> return new InMemoryMapOutput<K,V>(jobConf, mapId, this,
    (int)requestedSize, codec, primaryMapOutput)
>+> mapOutput.shuffle(host, is, compressedLength, decompressedLength, metrics, reporter)
    == InMemoryMapOutput.shuffle(host, is, ...)
    //跨节点从 Mapper 方拷贝文件内容到 InMemoryMapOutput 或 OnDiskMapOutput
>+> scheduler.copySucceeded(mapId, host, compressedLength, startTime, endTime, mapOutput)
    //告诉调度器,完成了从一个节点的 Map 输出文件拷贝
>+> remaining.remove(mapId) //这个 MapTask 的输出已经 shuffle 完毕
> } //end while (!remaining.isEmpty() && failedTasks == null)
> input.close()

```

这里的 mapOutput 是用来容纳 MapTask 输出文件 (Spill 文件) 的存储空间, 根据输出文件的大小和内存的使用情况, 这可以是 InMemoryMapOutput, 也可以是 OnDiskMapOutput。InMemoryMapOutput 需要预约, 因为 Fetcher 线程不止一个, 这会儿看到还有, 不加预约过一会儿说不定就没了。下面我们将只考察 InMemoryMapOutput, 把这搞明白, OnDiskMapOutput 也就容易理解了。

同一个目标节点上也许有不止一个 MapTask 已经完成运行, 所以先要进行一番整理, 从 maps 集合中取出所有属于同一 host 的 MapTask 的 TaskAttemptID。

然后, 实际的文件复制由 mapOutput.shuffle() 完成, 对于 InMemoryMapOutput 就是 InMemoryMapOutput.shuffle():

```
[Fetcher.run() > copyFromHost() > copyMapOutput() > InMemoryMapOutput.shuffle()]
```

```

InMemoryMapOutput.shuffle(MapHost host, InputStream input, long compressedLength,
    long decompressedLength, ShuffleClientMetrics metrics, Reporter reporter)
    //跨节点从 Mapper 方拷贝 Spill 文件
> checksumIn = new IFileInputStream(input, compressedLength, conf)
> input = checksumIn //增加了 checksum 环节后的输入流
> if (codec != null) { //如果有压缩, 就要增加解压缩环节
>+ decompressor.reset() //重启解压器模块
>+ input = codec.createInputStream(input, decompressor) //带上解压缩器的输入流
> }
> IOUtils.readFully(input, memory, 0, memory.length)
    //从 Mapper 方将特定 Partition 的数据读入 Reducer 方 InMemoryMapOutput 的缓冲区
> reporter.progress() //报告进度
> CodecPool.returnDecompressor(decompressor) //释放解压器

```

从对方把 Spill 文件中属于本 Partition 的数据复制过来之后, 回到上面 copyFromHost() 的代码中, 就通过 scheduler.copySucceeded() 告知 scheduler, 并将这个 MapTask 的 ID 从 remaining 集合中删除, 然后进入下一轮循环, 处理另一个 MapTask, 直至将当前已经完成运行的所有 MapTask 的 Spill 文件中属于这个 Partition 的数据都复制过来。

上面讲的是 Reducer 这一方 Fetcher 的操作, 它向对方即 Mapper 一方发出的是一条

Http GET 请求,就像我们在浏览器上访问一个网页、下载一个文件一样。与此相应,在 MapTask 所在的节点上就得有个 Http 的 Server 进程,就像一个网站一样。Http 网站的实现有很多种现成的软件可用,Hadoop 采用了开源的 Netty。Hadoop 的源代码中并不包含 Netty,而只是把它作为第三方软件拿来使用,对 Netty 的介绍也超出了本书的范围,所以我们在这里只要知道,Netty 服务器在得到来自 Fetcher 的 Http GET 请求之后,会回调由 Hadoop 提供的回调函数 ShuffleHandler.messageReceived(),这就行了。

```
ShuffleHandler.messageReceived(ChannelHandlerContext ctx, MessageEvent evt)
> HttpRequest request = (HttpRequest) evt.getMessage() //这是个 HTTP.GET 报文
> HttpResponse response = new DefaultHttpResponse(HTTP_1_1, OK) //准备好响应报文
> Map<String,List<String>> q = new QueryStringDecoder(request.getUri()).getParameters()
> List<String> mapIds = splitMaps(q.get("map")) //q 来自请求报文,因此 mapIds 也是
> ...
> Channel ch = evt.getChannel() // MessageEvent 中有通往客户端的连接和通道
> for (String mapId : mapIds) { //同一节点上也有可能多个 MapTask 存在
>+ MapOutputInfo info = mapOutputInfoMap.get(mapId)
//从 mapOutputInfoMap 中找到目标 Mapper 的输出信息 MapOutputInfo
>+ if (info == null) info = getMapOutputInfo(outputBasePathStr, mapId, reduceId, user)
//mapId 和 reduceId 均来自请求报文,reduceId 其实就是 Partition 的 Id
//注意,这个 info 是 MapOutputInfo,与下面 sendMapOutput() 中的不同
>+ lastMap = sendMapOutput(ctx, ch, user, mapId, reduceId, info)
>+> IndexRecord info = mapOutputInfo.indexRecord
//从目标 MapOutputInfo 中得到 IndexRecord,见 MapTask.sortAndSpill()
>+> ShuffleHeader header = new ShuffleHeader(mapId, info.partLength, info.rawLength, reduce)
//创建报头
>+> DataOutputBuffer dob = new DataOutputBuffer() //创建一个缓冲区
>+> header.write(dob) //把 ShuffleHeader 写入缓冲区 dob
>+> ch.write(wrappedBuffer(dob.getData(), 0, dob.getLength()))
>+> File spillfile = new File(mapOutputInfo.mapOutputFileName.toString()) //Spill 文件的路径
>+> RandomAccessFile spill = SecureIOUtils.openForRandomRead(spillfile, "r", user, null)
//打开这个 Spill 文件
>+> if (ch.getPipeline().get(SslHandler.class) == null) {
>+>+ FadviseFileRegion partition = new FadviseFileRegion(spill,
info.startOffset, info.partLength, manageOsCache, readaheadLength,
readaheadPool, spillfile.getAbsolutePath(),shuffleBufferSize,
shuffleTransferToAllowed)
>+>+ ChannelFuture writeFuture = ch.write(partition) //准备交给一个线程去异步发送
>+>+ writeFuture.addListener(new ChannelFutureListener() //要求在发送完成后得到通知
] operationComplete(ChannelFuture future) //发送完成后受回调
> if (future.isSuccess()) partition.transferSuccessful()
> partition.releaseExternalResources()
```

```
>+> return writeFuture //Netty 会将其交给一个线程去异步发送
> } //end for
```

这样,服务端即 Mapper 所在的节点会根据 mapId 找到其 Spill 文件,再根据 reduceId 即 Partition 的 Id 在 Spill 文件中找到属于该 Partition 的数据。所需的相关元数据在 IndexRecord 中,读者可以回头看一下 MapTask.sortAndSpill() 的摘要。找到这个 Mapper 的输出中属于这个 Partition 的数据之后,就将其交给 Netty,由其异步发送给客户端即 Reducer 这边的 Fetcher。

知道了跨节点的 Fetch,不妨再看一下 Uber 模式下的本地 Fetch,这是由 LocalFetcher 完成的。

```
class LocalFetcher<K,V> extends Fetcher<K, V> {}
] run()
  > maps = new HashSet<TaskAttemptID>()
  > for (TaskAttemptID map : localMapFiles.keySet()) maps.add(map)
  > while (maps.size() > 0) {
  >+ merger.waitForResource() //如有 merge 正在进行就等待其结束
  >+ doCopy(maps)
  >+> Iterator<TaskAttemptID> iter = maps.iterator()
  >+> while (iter.hasNext()) { //对于 maps 中的每一个 MapTask(其实一共就一个)
  >+>+ TaskAttemptID map = iter.next()
  >+>+ LOG.debug("LocalFetcher " + id + " going to fetch: " + map)
  >+>+ copyMapOutput(map)
  >+>+ if (...) iter.remove()
  >+>+ else break //We got back a WAIT command
                                // go back to the outer loop and block for InMemoryMerge
  >+> }
  > }
```

由此也可见,跨节点的 shuffle() 与本地的 doCopy() 是大致等价的。

Fetcher,不管是跨节点的 Fetcher,还是 LocalFetcher,总归是将 MapTask 所产生的 Spill 文件复制过来。不同的是,LocalFetcher 只需复制一个 Spill 文件,全部数据就到位了,因为在 Uber 模式下只有一个 MapTask;但是在常规的非 Uber 模式下会有很多 MapTask,那就要把所有 MapTask 的输出都复制过来,并加以合并排序,Reducer 的输入数据才算到位。所以下一步是合并。

10.10 Merge

Merge 的过程最好有个独立的上下文,即有个线程专管,这很好理解,所以 Hadoop 的代码中定义了一个抽象类:

```
abstract class MergeThread<T,K,V> extends Thread {}
```

```

] LinkedList<List<T>> pendingToBeMerged //待合并队列
] MergeManagerImpl<K,V> manager //所属的 MergeManagerImpl
] int mergeFactor //表示这个 merger 最多可做几路的合并
] startMerge(Set<T> inputs) //将数据源集合 inputs 挂入 pendingToBeMerged 队列
] run()

```

每个 MergeThread 都有自己的 pendingToBeMerged 队列,这个队列是 List 的 LinkedList,即队列的队列,队列中的一个元素就是一个 List,实际上就是一组输入数据源。想要让某个 MergeThread 线程合并一组数据源,就将这组数据源挂入它的这个队列。

不过,这个 MergeThread 只是个抽象类,需要有实体类加以落实。Hadoop 的代码中有三个类都是对 MergeThread 的落实和扩充:

```

class IntermediateMemoryToMemoryMerger
    extends MergeThread<InMemoryMapOutput<K, V>, K, V> {}
class InMemoryMerger extends MergeThread<InMemoryMapOutput<K,V>, K,V> {}
class OnDiskMerger extends MergeThread<CompressAwarePath,K,V> {}

```

这三个类都定义于 MergeManagerImpl 类内部,各有不同的用处。

我们在前面看到 ReduceTask.run()中创建了一个 MergeManagerImpl 对象。这个类的内部就定义了这三个实体类,并且在创建 MergeManagerImpl 对象的过程中至少创建了其中的两个,即 InMemoryMerger 线程和 OnDiskMerger 线程。下面是 MergeManagerImpl 类的摘要:

```

class MergeManagerImpl<K, V> implements MergeManager<K, V> {}
] DEFAULT_SHUFFLE_MEMORY_LIMIT_PERCENT = 0.25f
] MapOutputFile mapOutputFile
] Set<InMemoryMapOutput<K, V>> inMemoryMergedMapOutputs
] IntermediateMemoryToMemoryMerger memToMemMerger
] Set<InMemoryMapOutput<K, V>> inMemoryMapOutputs
] MergeThread<InMemoryMapOutput<K,V>, K,V> inMemoryMerger
] Set<CompressAwarePath> onDiskMapOutputs
] OnDiskMerger onDiskMerger
] Class<?extends Reducer> combinerClass
] CombineOutputCollector<K,V> combineCollector
] CompressionCodec codec
] MergeManagerImpl(TaskAttemptID reduceId, JobConf jobConf, FileSystem localFS, ...)
> ...
> this.memoryLimit = (long)
    (jobConf.getLong(MRJobConfig.REDUCE_MEMORY_TOTAL_BYTES,
        Math.min(Runtime.getRuntime().maxMemory(), Integer.MAX_VALUE))
        * maxInMemCopyUse)
> this.ioSortFactor = jobConf.getInt(MRJobConfig.IO_SORT_FACTOR, 100)

```

```

> boolean allowMemToMemMerge =
    jobConf.getBoolean(MRJobConfig.REDUCE_MEMTOMEM_ENABLED, false)
> if (allowMemToMemMerge) {
    // IntermediateMemoryToMemoryMerger 线程的创建是有条件的,可以配置
>+ this.memToMemMerger = new IntermediateMemoryToMemoryMerger(this,
    memToMemMergeOutputsThreshold)
>+ this.memToMemMerger.start()
> }
> this.inMemoryMerger = createInMemoryMerger() //创建 InMemoryMerger 线程
> this.inMemoryMerger.start() //并启动
> this.onDiskMerger = new OnDiskMerger(this) //创建 OnDiskMerger 线程
> this.onDiskMerger.start() //并启动
> this.mergePhase = mergePhase
] reserve(TaskAttemptID mapId, long requestedSize, int fetcher) //预留内存空间
] unconditionalReserve(TaskAttemptID mapId, long requestedSize, boolean primaryMapOutput)
] class InMemoryMerger extends MergeThread<InMemoryMapOutput<K,V>, K,V> {}
]] merge(List<InMemoryMapOutput<K,V>> inputs)
] class OnDiskMerger extends MergeThread<CompressAwarePath,K,V> {}
]] merge(List<CompressAwarePath> inputs)
] class IntermediateMemoryToMemoryMerger
    extends MergeThread<InMemoryMapOutput<K, V>, K, V> {}
]] merge(List<InMemoryMapOutput<K, V>> inputs)
] combineAndSpill(RawKeyValueIterator kvIter, Counters.Counter inCounter)
] class RawKVIteratorReader extends IFile.Reader<K,V> {}
] finalMerge(JobConf job, FileSystem fs, ...) //在 Shuffle.run()中调用
] class CompressAwarePath extends Path {}
] close()

```

可见,当创建 MergeManagerImpl 对象的时候,在其构造函数中至少就创建了 InMemoryMerger 和 OnDiskMerger 这两个类的线程。这两个类都是对 MergeThread 的扩充,又未定义自己的 run()函数,所以创建之后都是执行 MergeThread 的 run()函数。

MergeThread.run()

```

> while (true) {
>+ while(pendingToBeMerged.size() <= 0) pendingToBeMerged.wait()
    //等待,直至队列 pendingToBeMerged 非空
>+ inputs = pendingToBeMerged.removeFirst() //获取队列中的第一组输入
>+ merge(inputs) == InMemoryMerger.merge() 或 OnDiskMerger.merge()
> }

```

很简单,就是监视着自己的 pendingToBeMerged 队列,一旦这个队列非空,就从中摘下一组输入源,调用 merge()加以合并。MergeThread 的 merge()是个抽象函数,所以实际调用的

是 `InMemoryMerger.merge()` 或 `OnDiskMerger.merge()`。

那么这个 `pendingToBeMerged` 队列中的一组组数据源是怎么进来的呢? `MergeThread` 的 `startMerge()` 就是干这个用的, 当完成了 `Spill` 文件的复制而关闭复制过来的文件的时候, 就会在例如 `closeInMemoryFile()` 中调用这个函数:

```
MergeThread.startMerge(Set<T> inputs)
> numPending.incrementAndGet()
> toMergeInputs = new ArrayList<T>()
> Iterator<T> iter = inputs.iterator() //inputs 是个序列, 这个序列的每个元素
> for (int ctr = 0; iter.hasNext() && ctr < mergeFactor; ++ctr) {
>+ toMergeInputs.add(iter.next()) //从 inputs 收集需要合并的输入, 不超过 mergeFactor 个
//也就是说, 这个线程只能进行最多 mergeFactor 路的合并
>+ iter.remove()
> }
> pendingToBeMerged.addLast(toMergeInputs) //将一组需要合并的输入挂入队列
> pendingToBeMerged.notifyAll() //然后发出通知, 唤醒相关的线程
```

要让一个 `MergeThread` 对一组数据 `input` 合并排序, 就调用其 `startMerge()` 函数, 将这组数据挂入其 `pendingToBeMerged` 队列。注意, 这里所说的一组数据, 是指一组数据源, 而不是单项数据。程序中的 `for` 循环, 是对各项数据源的循环。打个比方, 假定有 5 个排好序的数据文件需要加以合并, 那么这个 `inputs` 集合中就有 5 个元素, 每个元素代表着一个数据文件, 这里的 `for` 循环就循环 5 次。至于具体数据文件中的数据量, 那可就大了。这个 `for` 循环的次数有个限制, 就是不能大于 `mergeFactor`, 这是在创建线程时作为参数传下来的, 表示最多可做多少路的合并。

如前所述, `MergeThread` 是个抽象类, 具体则有 `InMemoryMerger` 和 `OnDiskMerger`。另外还有一种 `IntermediateMemoryToMemoryMerger`, 相对就没有那么重要了。显然, 需要合并的数据在内存中还是在磁盘上, 具体的合并方法也就有所不同。相比之下, 应该是在内存中的操作比较简单也比较快, 所以我们总是希望能把数据都放在内存中, 但是数据量大了就只好写到磁盘上去了。在 Hadoop 的 MapReduce 这个框架中, `MapTask` 的输出并非“细水长流”源源不断地到达 `ReduceTask` 这一边, 而是在 `Map` 阶段结束时一下子涌来的, 此时最大的可能就是只好写到磁盘上。这样, 在数据量很大的情况下, `Mapper` 的输出不仅在 `MapTask` 那一边就得因 `Spill` 写一次盘, 并且在合并 `Spill` 文件时又得写一次, 而且在 `ReduceTask` 这一边也还得再写一次。这当然会显著影响数据处理的效率。不过事情也不绝对, 如果数据量不大而无须 `Spill`, 在 `ReduceTask` 这一边也无须采用 `OnDiskMerger`, 那就又不一样了。

不过, 无论是 `InMemory` 还是 `OnDisk`, `Merge` 操作的原理还是一样的, 所以我们在这里只关注 `InMemory` 的 `Merge`, 即 `InMemoryMerger.merge()`。

```
[MergeThread.run() > InMemoryMerger.merge()]
```

```
InMemoryMerger.merge(List<InMemoryMapOutput<K,V>> inputs)
> TaskAttemptID mapId = inputs.get(0).getMapId() //获取其中第一个输入源
```

```

> TaskID mapTaskId = mapId.getTaskID()           //属于哪一个任务
> inMemorySegments = new ArrayList<Segment<K, V>>()
> mergeOutputSize = createInMemorySegments(inputs, inMemorySegments, 0)
    //根据输入列表 inputs, 构建一个用于合并的 Segment 列表
> noInMemorySegments = inMemorySegments.size()   //共有几个 Segment
> outputPath = mapOutputFile.getInputFileForWrite(mapTaskId, mergeOutputSize).
    suffix(Task.MERGED_OUTPUT_PREFIX)
    //以第一个输入源(Mapper)的 TaskId 为主构成输出文件名, 扩充名为“.merged”
> FSDataOutputStream out = CryptoUtils.wrapIfNecessary(jobConf, rfs.create(outputPath))
    //在本地文件系统中创建输出文件
> Writer<K, V> writer = new Writer<K, V>(jobConf, out,
    (Class<K>) jobConf.getMapOutputKeyClass(),
    (Class<V>) jobConf.getMapOutputValueClass(), codec, null, true)
    //创建针对 K、V 数据类型的 Writer, 也许需要压缩
> RawKeyValueIterator rIter =
    Merger.merge(jobConf, rfs, (Class<K>) jobConf.getMapOutputKeyClass(),
    (Class<V>) jobConf.getMapOutputValueClass(),
    inMemorySegments, inMemorySegments.size(),
    new Path(reduceId.toString()),
    (RawComparator<K>) jobConf.getOutputKeyComparator(),
    reporter, spilledRecordsCounter, null, null)
    //合并 inMemorySegments 列表中的各个 Segment
    //这个 Merger 就是 hadoop.mapred.Merger
> if (null == combinerClass) { //如果没有安排 Combine, 就将结果直接写入文件
>+ Merger.writeFile(rIter, writer, reporter, jobConf)
    //writer 会通过输出流 out 把排好序的数据写入 outputPath, 即扩充名为“.merged”的文件
> } else { //如果安排了 Combine 环节, 就要经过 Combine 环节
>+ combineCollector.setWriter(writer)           //在 writer 中插入 combiner
>+ combineAndSpill(rIter, reduceCombineInputCounter)
> }
> compressAwarePath = new CompressAwarePath(outputPath,
    writer.getRawLength(), writer.getCompressedLength())
> closeOnDiskFile(compressAwarePath) //关闭输出文件
>> onDiskMapOutputs.add(file) //将输出文件加入 onDiskMapOutputs 集合
>> if (onDiskMapOutputs.size() >= (2 * ioSortFactor - 1)) {
>>+ onDiskMerger.startMerge(onDiskMapOutputs)
    //太大了, 单靠 InMemoryMerger 解决不了, 只好请 OnDiskMerger 帮忙
>> }

```

最终的合并还是要通过 `Merger.merge()` 完成。这个 `Marger` 与 `MapTask` 中用来合并 `Spill` 文件的是同一个 `Merger`, 即 `org.apache.hadoop.mapred.Merger`, 那是对具体合并算法的

实现,属于数据结构和算法方面的问题,我们就不细看了。

Merge 和 Combine 这两个词都有“合并、结合”的意思,这二者又有什么区别呢? Merge 是多个输入流的合并,Reducer 的输入是来自多个不同的 Mapper,在 reduce()之前要把多个输入流合并成一个,这是横向的合并。而 Combine,则是在经过横向合并之后的输入流中再做纵向合并,因为横向合并所形成的输入流中很可能前后之间又可以合并。其实这种纵向合并完全可以看作是 reduce()的一部分,有无 Combine 这个环节只是要不要把 Reduce 这个工作分两步做的问题,所以 Reducer 通常也用来充当 CombinerClass。

如果安排了 Combiner 这个环节,那就跟 MapperRunner 和 ReducerRunner 一样也要有个 CombinerRunner。在 Hadoop 的代码中,有新老两种 CombinerRunner,都是对抽象类 CombinerRunner 的扩充。要创建一个 CombinerRunner 时,可根据具体的系统配置创建新的或旧的 CombinerRunner:

```
abstract class CombinerRunner<K,V>{
] create(JobConf job, TaskAttemptID taskId, Counters.Counter inputCounter, ...)
    > cls = (Class<?extends Reducer<K,V,K,V>>) job.getCombinerClass()
    > if (cls!= null) return new OldCombinerRunner(cls, job, inputCounter, reporter)
    > taskContext = new TaskAttemptContextImpl(job, taskId, reporter)
    > newcls = taskContext.getCombinerClass()
    > if (newcls!= null) return new NewCombinerRunner<K,V>(
                                                newcls, job, taskId, taskContext, ...)
    > return null
}
```

在老版本的 Hadoop 中,关于 CombinerClass 的设置放在 JobConf 中,在新版本中则把这个信息移到了 TaskAttemptContextImpl 中。我们假定采用 NewCombinerRunner:

```
class NewCombinerRunner<K, V> extends CombinerRunner<K,V> {}
] NewCombinerRunner(Class reducerClass, JobConf job, ...) //构造函数
    > this.reducerClass = reducerClass //其实是 combinerClass
    > keyClass = (Class<K>) context.getMapOutputKeyClass()
    > valueClass = (Class<V>) context.getMapOutputValueClass()
    > comparator = (RawComparator<K>) context.getCombinerKeyGroupingComparator()
    > this.committer = committer
] class OutputConverter<K,V> extends org.apache.hadoop.mapreduce.RecordWriter<K,V> {}
] combine(RawKeyValueIterator iterator, OutputCollector<K,V> collector)
    > reducer = (org.apache.hadoop.mapreduce.Reducer<K,V,K,V>)
                                ReflectionUtils.newInstance(reducerClass, job)
                                //创建 combiner,但是实际上就是一种 Reducer
    > reducerContext = createReduceContext(reducer, job, taskId, ...)
    > reducer.run(reducerContext) //启动 combiner
```

注意,构造函数 NewCombinerRunner() 的第一个调用参数 reducerClass,这就是上面 CombinerRunner.create() 中的实参 newcls。这本应是个 combinerClass,这里却说是 reducerClass,

就是因为这二者其实没有什么区别,只是把一件事分两步做而已,所以我们对这里的 Combine 环节不必太在意。

就这样,在 Shuffle 的过程中,一旦把一些 MapTask 产生的 Spill 文件复制了过来,就将其挂入某个合并线程的 pendingToBeMerged 队列,这个线程就会加以合并,合并后的输出是扩充名为“.merged”的文件。

最后,在 Shuffle.run()中,在所有 MapTask 的输出文件都已复制过来并加以合并之后,调用了 merger.close(),那就是 MergeManagerImpl.close(),为 Shuffle 和合并的阶段画上句号,再通过 finalMerge()把所有的.merged 文件都合并在一起:

```
[Shuffle.run() > MergeManagerImpl.close()]
```

```
MergeManagerImpl.close() //Finish the on-going merges...
> // Wait for on-going merges to complete
> if (memToMemMerger != null) {
>+ memToMemMerger.close()
> }
> inMemoryMerger.close() == MergeThread.close()
>> closed = true
>> waitForMerge() //等待,直至该线程的队列中不再有需要合并的数据源
>>> while (numPending.get() > 0) wait()
>> interrupt() //中止线程的运行
> onDiskMerger.close() == MergeThread.close()
> List<InMemoryMapOutput<K, V>> memory =
    new ArrayList<InMemoryMapOutput<K, V>> (inMemoryMergedMapOutputs)
    //创建一个 InMemoryMapOutput 的集合 memory
> inMemoryMergedMapOutputs.clear() //清除 inMemoryMergedMapOutputs 集合的内容
> memory.addAll(inMemoryMapOutputs) //把 inMemoryMapOutputs 集合的内容转移过去
> inMemoryMapOutputs.clear()
> List<CompressAwarePath> disk = new ArrayList<CompressAwarePath> (onDiskMapOutputs)
    //磁盘上.merged 文件的集合
> onDiskMapOutputs.clear() //清除 onDiskMapOutputs 集合的内容
> return finalMerge(jobConf, rfs, memory, disk) //合并 memory 和 disk 两个集合中的数据源
```

为把所有数据源都合并起来,先把内存中的数据集合都整理在 memory 集合中,把磁盘上的数据集合都整理在 disk 集合中,然后通过 finalMerge()把它们合并在一起。

```
[Shuffle.run() > MergeManagerImpl.close() > finalMerge()]
```

```
10 finalMerge(JobConf job, FileSystem fs,
    List<InMemoryMapOutput<K, V>> inMemoryMapOutputs,
    List<CompressAwarePath> onDiskMapOutputs)
    > maxRedPer = job.getFloat(MRJobConfig.REDUCE_INPUT_BUFFER_PERCENT, 0f)
```

```

> maxInMemReduce = (int)Math.min(Runtime.getRuntime().maxMemory() * maxRedPer,
                                     Integer.MAX_VALUE)
> Class<K> keyClass = (Class<K>)job.getMapOutputKeyClass()
> Class<V> valueClass = (Class<V>)job.getMapOutputValueClass()
> boolean keepInputs = job.getKeepFailedTaskFiles()
> Path tmpDir = new Path(reduceId.toString())
> comparator = (RawComparator<K>)job.getOutputKeyComparator()
> memDiskSegments = new ArrayList<Segment<K,V>>()
> inMemToDiskBytes = 0
> if (inMemoryMapOutputs.size() > 0) {
>+ mapId = inMemoryMapOutputs.get(0).getMapId().getTaskID()
>+ inMemToDiskBytes = createInMemorySegments(inMemoryMapOutputs,
                                             memDiskSegments, maxInMemReduce)
>+ numMemDiskSegments = memDiskSegments.size()
>+ if (numMemDiskSegments > 0 && ioSortFactor > onDiskMapOutputs.size()) {
>++ mergePhaseFinished = true
>++ Path outputPath = mapOutputFile.getInputFileForWrite(mapId, inMemToDiskBytes).
                                     suffix(Task.MERGED_OUTPUT_PREFIX)
>++ rIter = Merger.merge(job, fs, keyClass, valueClass, memDiskSegments, ...)
>++ FSDataOutputStream out = CryptoUtils.wrapIfNecessary(job, fs.create(outputPath))
>++ writer = new Writer<K, V>(job, out, keyClass, valueClass, codec, null, true)
>++ Merger.writeFile(rIter, writer, reporter, job)
>++ writer.close()
>++ onDiskMapOutputs.add(new CompressAwarePath(outputPath,
                                             writer.getRawLength(), writer.getCompressedLength()))
>++ LOG.info("Merged " + numMemDiskSegments + " segments, " +
            inMemToDiskBytes + " bytes to disk to satisfy " + "reduce memory limit")
>++ inMemToDiskBytes = 0
>++ memDiskSegments.clear()
>+ } else if (inMemToDiskBytes != 0) {
>++ LOG.info("Keeping " + numMemDiskSegments + " segments, " +
            inMemToDiskBytes + " bytes in memory for " + "intermediate, on-disk merge")
>+ }
> }
> // segments on disk
> diskSegments = new ArrayList<Segment<K,V>>()
> onDiskBytes = inMemToDiskBytes
> rawBytes = inMemToDiskBytes
> onDisk = onDiskMapOutputs.toArray(new CompressAwarePath[onDiskMapOutputs.size()])
> for (CompressAwarePath file : onDisk) {

```

```

>+ fileLength = fs.getFileStatus(file).getLen()
>+ onDiskBytes += fileLength
>+ rawBytes += (file.getRawDataLength() > 0)?file.getRawDataLength() : fileLength
>+ LOG.debug("Disk file: " + file + " Length is " + fileLength)
>+ diskSegments.add(new Segment<K, V>(job, fs, file, codec, keepInputs, ...))
> }
> LOG.info("Merging " + onDisk.length + " files, " + onDiskBytes + " bytes from disk")
> comparator = new Comparator<Segment<K,V>>() {
    ] compare(Segment<K, V> o1, Segment<K, V> o2)
        > if (o1.getLength() == o2.getLength()) return 0
        > return o1.getLength() < o2.getLength()? -1 : 1
> }
> Collections.sort(diskSegments, comparator)
> // build final list of segments from merged backed by disk + in - mem
> finalSegments = new ArrayList<Segment<K,V>>()
> inMemBytes = createInMemorySegments(inMemoryMapOutputs, finalSegments, 0)
> LOG.info("Merging " + finalSegments.size() + " segments, " +
            inMemBytes + " bytes from memory into reduce")
> if (0 != onDiskBytes) { //如果有数据在磁盘上
>+ numInMemSegments = memDiskSegments.size()
>+ diskSegments.addAll(0, memDiskSegments)
>+ memDiskSegments.clear()
>+ // Pass mergePhase only if there is a going to be intermediate merges.
    // See comment where mergePhaseFinished is being set
>+ Progress thisPhase = (mergePhaseFinished)?null : mergePhase
>+ RawKeyValueIterator diskMerge = Merger.merge(job, fs, keyClass, valueClass, codec, ...)
>+ diskSegments.clear()
>+ if (0 == finalSegments.size()) return diskMerge
>+ reader = new RawKVIteratorReader(diskMerge, onDiskBytes, true, rawBytes))
>+ finalSegments.add(new Segment<K,V>(reader, true, rawBytes))
> }
> return Merger.merge(job, fs, keyClass, valueClass, finalSegments, finalSegments.size(),
            tmpDir, comparator, reporter, spilledRecordsCounter, null, null)

```

最后,finalMerge()的返回值就是Merger.merge()的返回值,这个返回值是一个 RawKeyValueIterator,也就是一个 RawKeyValue 的序列。这个序列将被用作 Reducer 的输入。

10.11 Reduce 阶段的输入和输出

在前面的 merge 阶段中,最后 merger.close(),即 MergeManagerImpl.close()返回的是一个 RawKeyValueIterator,我们不妨细看一下 Shuffle.run()中的这几行语句:


```

RawKeyValueIterator run() throws IOException, InterruptedException {
    ...
    // Finish the on-going merges...
    RawKeyValueIterator kvIter = null;
    try {
        kvIter = merger.close();
    } catch (Throwable e) {
        throw new ShuffleError("Error while doing final merge ", e);
    }
    ...
    return kvIter;
}

```

所谓 Iterator 其实就是一个序列, 一个供 for 循环或 while 循环逐个元素加以处理的序列。源自诸多 Mapper 输出的大量 KV 对, 经过一系列的 Sort、Combine, 特别是 Merge 以后就形成了一个线性的序列, 这就是这里的 RawKeyValueIterator, 严格地说是实现了 RawKeyValueIterator 界面的某种序列。Hadoop 的代码中实现了这个界面的类有 SequenceFile.MergeQueue、MapTask.MRResultIterator、MapTask.MRResultIterator 以及 Merger.MergeQueue, 在这里显然是 Merger.MergeQueue。正是这个序列, 在 Reduce 阶段被用作 Reducer 的输入。不过, 此时这个序列中的元素不再是简单的 KV 对, 而是每个 K 后面有一串 V。

所以, Reducer 的 InputFormat 不像 Mapper 的 InputFormat 那样多样, 而是形式上相当整齐, 因为它的输入文件本来就是在 Hadoop 内部生成的, 只是 K 和 V 的类型不同而已。注意, Reducer 输出端的 K 和 V 有可能跟 Mapper 输入端的 K 和 V 不是相同的类型、相同的语义, 需要细加分辨。

Reducer 输出的去处, 最常见的当然是文件, 所以有关文件的 OutputFormat 显然最重要、最常用。然而, 对于文件的 RecordWriter 却是最简单的, 对本书的读者实无必要再加细述。

另一个可能的去处是数据库, 尤其是关系式数据库, 这倒可以作为一个例子看一下。

关于 OutputFormat, 源码中 mapreduce 与 mapred 两个分支上有不同的定义和实现, 我们现在看的是 mapreduce 分支。在这个分支上, 先定义了一个抽象类 OutputFormat<K, V>:

```

abstract class OutputFormat<K, V> {}
] abstract RecordWriter<K, V> getRecordWriter(TaskAttemptContext context)
] abstract void checkOutputSpecs(JobContext context)
] abstract OutputCommitter getOutputCommitter(TaskAttemptContext context)

```

这个抽象类其实与界面无异, 它的三个函数全是抽象函数。

在此基础上 Hadoop 定义和提供了不少的输出格式:

```

abstract class FileOutputFormat<K, V> extends OutputFormat<K, V> {}
    class SequenceFileOutputFormat <K,V> extends FileOutputFormat<K, V> {}
    class SequenceFileAsBinaryOutputFormat

```

```

        extends SequenceFileOutputFormat <BytesWritable,BytesWritable> {}
    class TextOutputFormat<K, V> extends FileOutputFormat<K, V> {}
        class CopyOutputFormat<K, V> extends TextOutputFormat<K, V> {
    class MapFileOutputFormat extends FileOutputFormat<WritableComparable, Writable> {}
class FilterOutputFormat<K, V> implements OutputFormat<K, V> {}
    class LazyOutputFormat<K, V> extends FilterOutputFormat<K, V> {}
class DBOutputFormat<K extends DBWritable, V> extends OutputFormat<K,V> {}
class FilterOutputFormat <K,V> extends OutputFormat<K, V> {}
class NullOutputFormat<K, V> extends OutputFormat<K, V> {}

```

另外还有个特殊的 class MultipleOutputs {},但这却不是对任何输出格式的扩充。

我们从上面所列的诸多输出格式中选择 DBOutputFormat 为例来说明 Reducer 对于 OutputFormat 和相应 RecordWriter 的使用。DBOutputFormat 的摘要如下:

```

class DBOutputFormat<K extends DBWritable, V> extends OutputFormat<K,V> {}
] getOutputCommitter(TaskAttemptContext context)
    > return new FileOutputCommitter(FileOutputFormat.getOutputPath(context), context)
] constructQuery(String table, String[] fieldNames)
] getRecordWriter(TaskAttemptContext context)
] setOutput(Job job, String tableName, String... fieldNames)
] class DBRecordWriter extends RecordWriter<K, V> {}
]] Connection connection
]] PreparedStatement statement
]] DBRecordWriter(Connection connection, PreparedStatement statement)
    > this.connection = connection
    > this.statement = statement
    > this.connection.setAutoCommit(false)
]] write(K key, V value)
    > key.write(statement) == DBCountPageView.PageviewRecord.write()
    > statement.addBatch()
]] close(TaskAttemptContext context)
    > statement.executeBatch()
    > connection.commit()

```

Hadoop 源码中有个示例 DBCountPageView,就演示了怎样使用这个 DBOutputFormat。我们先看一下应用软件中有关 OutputFormat 的设置:

```

class DBCountPageView extends Configured implements Tool {}
] run(String[] args)
    > String driverClassName = DRIVER_CLASS // "org.hsqldb.jdbc.JDBCdriver"
    > String url = DB_URL // "jdbc:hsqldb:hsql://localhost/URLAccess"
    > initialize(driverClassName, url) //对数据库
    >> if(driverClassName.equals(DRIVER_CLASS)) {

```

```

>>>+ startHsqldbServer()           //启动数据库服务器
>>> }
>>> createConnection(driverClassName, url) //建立与数据库服务器的连接
>>> dropTables()
>>>> String dropAccess = "DROP TABLE Access" //用来撤销原有的 Access 表
>>>> String dropPageview = "DROP TABLE Pageview" //用来撤销原有的 Pageview 表
>>>> Statement st = connection.createStatement() //创建数据库查询
>>>> st.executeUpdate(dropAccess) //撤销数据库中的 Access 表
>>>> st.executeUpdate(dropPageview) //撤销数据库中的 Pageview 表
>>>> connection.commit() //确认操作,从数据库中删除原有的两个表
>>>> st.close()
>>> createTables() //在数据库中新建 Access 和 Pageview 两个表,为 MR 的输出做好准备
>>> populateAccess() //往 Access 表中插入若干记录
> DBConfiguration.configureDB(conf, driverClassName, url)
> Job job = new Job(conf)
> job.setMapperClass(PageviewMapper.class)
> job.setCombinerClass(LongSumReducer.class)
> job.setReducerClass(PageviewReducer.class)
> ...
> DBInputFormat.setInput(job, AccessRecord.class, "Access", null, "url", AccessFieldNames)
> DBOutputFormat.setOutput(job, "Pageview", PageviewFieldNames)
>>> DBConfiguration dbConf = setOutput(job, tableName)
>>>> job.setOutputFormatClass(DBOutputFormat.class) //设置作业的输出格式为 DB
>>>> job.setReduceSpeculativeExecution(false)
>>>> dbConf = new DBConfiguration(job.getConfiguration())
>>>> dbConf.setOutputTableName(tableName)
>>>> return dbConf
>>> dbConf.setOutputFieldNames(fieldNames)
> ...
> job.setOutputKeyClass(PageviewRecord.class) //reducer 输出 K 的类型
> job.setOutputValueClass(NullWritable.class) //reducer 输出 V 的类型
> ...
> ret = job.waitForCompletion(true)?0:1

```

我们对这个例子本身并无兴趣,关心的是如何将其 Reducer 的输出写入一个关系式数据库,以此作为 MR 数据流的最后一站的一个实例。

这样,到投运这个作业的 ReduceTask 的时候,在 runNewReducer() 中创建的就是 DBOutputFormat 的 RecordWriter,即 DBRecordWriter。

```
[ReduceTask.run() > runNewReducer()]
```

```

runNewReducer(JobConf job, TaskUmbilicalProtocol umbilical, TaskReporter reporter,
    RawKeyValueIterator rIter, RawComparator<INKEY> comparator,
    Class<INKEY> keyClass, Class<INVALUE> valueClass)
> ...
> TaskAttemptContext taskContext = new TaskAttemptContextImpl(job, getTaskID(), reporter)
> ...
> RecordWriter<OUTKEY,OUTVALUE> trackedRW =
    new NewTrackingRecordWriter<OUTKEY, OUTVALUE>(this, taskContext)
>> this.outputRecordCounter = reduce.reduceOutputCounter //这是在其构造方法中
>> this.fileOutputByteCounter = reduce.fileOutputByteCounter
>> ...
>> this.real = (org.apache.hadoop.mapreduce.RecordWriter<K, V>)
    reduce.outputFormat.getRecordWriter(taskContext)
    == DBOutputFormat.getRecordWriter(taskContext) //创建 RecordWriter
>>> dbConf = new DBConfiguration(context.getConfiguration())
>>> tableName = dbConf.getOutputTableName() //数据库中表的名称
>>>> return conf.get(DBConfiguration.OUTPUT_TABLE_NAME_PROPERTY)
>>> String[] fieldNames = dbConf.getOutputFieldNames() //有关字段的名称
>>>> return conf.getStrings(DBConfiguration.OUTPUT_FIELD_NAMES_PROPERTY)
>>> Connection connection = dbConf.getConnection() //建立数据库连接
>>>> Class.forName(conf.get(DBConfiguration.DRIVER_CLASS_PROPERTY))
>>>> if(conf.get(DBConfiguration.USERNAME_PROPERTY) == null) {
>>>>+ return DriverManager.getConnection(conf.get(DBConfiguration.URL_PROPERTY))
>>>> } else {
>>>>+ return DriverManager.getConnection(
    conf.get(DBConfiguration.URL_PROPERTY), //数据库的 URL
    conf.get(DBConfiguration.USERNAME_PROPERTY), //登录用户名
    conf.get(DBConfiguration.PASSWORD_PROPERTY)) //口令
    //建立跟数据库 Server 的连接
>>>> }
>>> query = constructQuery(tableName, fieldNames) //构建用于 SQL 语句的字符串
>>> statement = connection.prepareStatement(query) //进一步构建 SQL 语句
>>> return new DBRecordWriter(connection, statement)
    //创建 DBRecordWriter
    //返回后被进一步包装成 NewTrackingRecordWriter,就是前面的 trackedRW
>> ...
> Reducer.Context reducerContext = createReduceContext(reducer, job, getTaskID(), rIter,
    reduceInputKeyCounter, reduceInputValueCounter, trackedRW,
    committer, reporter, comparator, keyClass, valueClass)
>> reduceContext = new ReduceContextImpl<INKEY, INVALUE, OUTKEY, OUTVALUE>(job,

```

```

        taskId, rIter, inputKeyCounter, inputValueCounter,
        output, committer, reporter, comparator, keyClass, valueClass)
        //这个 output 就是 trackedRW
>>> super(conf, taskId, output, committer, reporter)
        == TaskInputOutputContextImpl(conf, taskId, output, committer, reporter)
>>>> this.output = output    //这就是 RecordWriter
>> reducerContext =
        new WrappedReducer<INKEY, INVALUE, OUTKEY, OUTVALUE>()
        .getReducerContext(reduceContext)
>> return reducerContext
        //创建 ReduceContextImpl 对象,这就是 reducer.run()中的那个 context
> reducer.run(reducerContext) == Reducer.run(reducerContext)
>> setup(context)
>> while (context.nextKey()) {
>>+ reduce(context.getCurrentKey(), context.getValues(), context) == Reducer.reduce()
>>+> for(VALUEIN value: values) {
>>+>+ context.write((KEYOUT) key, (VALUEOUT) value)
        == ReduceContextImpl.write(key, value) //从 TaskInputOutputContextImpl 继承
        == TaskInputOutputContextImpl.write(key, value)
>>+>+> output.write(key, value) => DBRecordWriter.write(key, value)
>>+> }
>>+ Iterator<VALUEIN> iter = context.getValues().iterator()
>>+ if(iter instanceof ReduceContext.ValueIterator) {
>>++ ((ReduceContext.ValueIterator<VALUEIN>)iter).resetBackupStore()
>>+ }
>> }
>> cleanup(context)

```

我们特别关注的是 Reducer 运行时其 context 中的 output 究竟是什么,因为这是个 RecordWriter,Reducer 的输出就是由它写出去的。这里我们看到,由于 OutputFormat 是 DBOutputFormat,这个 RecordWriter 就是基于 DBRecordWriter 的 NewTrackingRecordWriter,这就是程序中的 trackedRW。这个 trackedRW 被用作调用 createReduceContext()时的参数之一,最后被设置成 TaskInputOutputContextImpl,即 ReduceContextImpl(后者扩充了前者)的 output。这就为 Reducer 的运行做好了最根本的准备,把数据流的最后一个环节搭建好了。这里之所以要把 DBRecordWriter 包装成 NewTrackingRecordWriter,是为了这样就可以 Tracking,可以跟踪程序的执行。

在开始运行 Reducer 之前,先要建立好跟数据库服务器的连接,这就像先打开文件一样。但是,建立数据库连接当然比打开文件要复杂一些。除了建立连接之外,这里还为后面要用到的 INSERT 语句做些准备,其中的 constructQuery()构建一个 SQL“查询”短语。虽然对数据库的操作是写入,是 INSERT,但大家都已习惯使用“查询”这个词:

```
[runNewReducer() > NewTrackingRecordWriter() > DBOutputFormat.getRecordWriter() >
DBOutputFormat.constructQuery()]
```

```
DBOutputFormat.constructQuery(String table, String[] fieldNames)
> StringBuilder query = new StringBuilder()
> query.append("INSERT INTO ").append(table)
> if (fieldNames.length > 0 && fieldNames[0] != null) {
>+ query.append(" (")
>+ for (int i = 0; i < fieldNames.length; i++) {
>++ query.append(fieldNames[i])
>++ if (i != fieldNames.length - 1) query.append(",")
>+ }
>+ query.append(")")
> }
> query.append(" VALUES (")
> for (int i = 0; i < fieldNames.length; i++) {
>+ query.append("?")
>+ if (i != fieldNames.length - 1) query.append(",")
> }
> query.append(");")
> return query.toString()
```

从这段代码摘要可以看出,所形成的字符串大致如下:

```
"INSERT INTO tablename (field1, field2, ...) VALUES(?,?,...)"
```

然后以此字符串为基础构成一个针对具体数据库的 SQL 语句。再在所建立的数据库连接和 SQL 语句的基础上创建 DBRecordWriter 对象。

回到 runNewReducer(),做好了上述这些准备之后,把这些“部件”都收集在一起构建一个 reducerContext,实际上是个 ReduceContextImpl 对象,就可以调用 Reducer.run()了。

DBCountPageView 这个 App 的 Reducer 本是 PageviewReducer,这是对 Reducer 的扩充,但是一般 App 提供的 reducer 不应提供自己的 run()来覆盖 Reducer.run(),而只应提供自己的 reduce()函数,所以这里调用的仍是 Reducer.run(),但是里面调用的 reduce()则是 PageviewReducer.reduce()了。

```
[runNewReducer() > Reducer.run() > PageviewReducer.reduce()]
```

```
PageviewReducer.reduce(Text key, Iterable<LongWritable> values, Context context)
> long sum = 0L
> for(LongWritable value: values) {
>+ sum += value.get()
> }
> r = new PageviewRecord(key.toString(), sum)
```



```
> context.write(r, n) == ReduceContextImpl.write(r, n)
== TaskInputOutputContextImpl.write(KEYOUT key, VALUEOUT value)
>> output.write(key, value) == NewTrackingRecordWriter.write(K key, V value)
>>> real.write(key, value) == DBRecordWriter.write(key, value)
>>>> key.write(statement) == PageviewRecord.write(statement)
>>>>> statement.addBatch()    //这是由 java.sql 提供的批处理,把语句加到批处理 list 中
```

可见,由于把 `OutputFormat` 设置成了 `DBOutputFormat`,这个 `Reducer` 的输出是通过 `DBRecordWriter` 写出去的,而 `DBRecordWriter` 则把输出插入数据库中预定的表中。

就像 `InputFormat` 与 `RecordReader` 一样,每种 `OutputFormat` 也都有自己的 `RecordWriter`,在 App 中为作业设置了什么样的 `OutputFormat`,`Reducer` 在输出时就会采用什么样的 `RecordWriter`。当然,最常用的 `OutputFormat` 还是几种 `FileOutputFormat`。

第11章

Hadoop 的文件系统 HDFS

11.1 文件的分布与容错

HDFS 是 Hadoop 集群的文件系统,这是一种分布(distributed)、容错(fault tolerant)的文件系统。注意,我们用“HDFS”这个词表示的是 Hadoop 的两大子系统之一,这并非一个类的名称。Hadoop 的代码中倒是定义了一个类叫 Hdfs,但其实并未看到对这个类的引用。作为 HDFS 的门户,在 App 面前代表着 HDFS 的倒是 DistributedFileSystem。

所谓分布,是说整个文件系统的内容并非集中存储在一台或几台“文件服务器上”,而是分散在集群的不同节点上。理想的情景是集群内的每一台机器都承担着一些内容的存储。这跟一般采用文件服务器和 SAN(存储域网)或 LAN(局域网)的结构形态形成鲜明的对比。确实,既然集群中的每一个节点都是商品 PC 机,每台机器都配有容量不算很小的磁盘,为什么不把它们利用起来呢?

以前,人们在讲到分布式文件系统的时候总是着眼于容量和容错的问题,但是现在有了大数据处理的要求,情况就又有所不同了。对于大数据处理系统,文件系统之所以应该是分布式的,不再仅仅是因为容量和容错的问题。大数据处理需要并行化,需要把计算量分摊到很多节点上,但是,如果海量的数据都集中在一起,需要在计算的时候通过网络传输过去,那就不大现实了,因为这个集中点及其周边的网络显然会成为瓶颈。所以大数据处理有个原则,就是数据在哪里,计算就在哪里。可想而知,在这样的条件下,分布的计算必然要求分布的数据存储,最好是每个节点都存储数据,每个节点也都承担计算。

但是,按什么方式把整个文件系统的内容分布存储在集群中,则是个值得研究的问题。

人们早就掌握了远程挂载(mount)文件系统的技术,通过网络可以把别的机器上的某个文件目录挂载到本地,成为本地的一个子目录。设想,如果集群内每台机器上都提供一个供远程挂载的目录,同时也把其他节点上的此类目录挂载到本地,使得例如一个子目录的内容在 A 节点上,另一个子目录的内容在 B 节点上,这样是不是就实现了整个文件系统的分布?恐怕不能说这样的方式就不算分布,但这只是一种宏观的、大粒度的分布,这只是文件目录(文件系统)层面的分布。这种方式跟文件服务器其实没有很大区别。设想,如果某个子目录中的文件在某一时间段成了热点,集群中几乎所有的节点都要来访问承载着这个子目录的节点,这个节点就成了整个集群的瓶颈。

那么我们把粒度细化一点,改成文件层面的分布。也就是说,不是以目录为单位,而是以文件为单位的分布。每个节点承载着若干(可能是几千、几万)个文件的存储,但是这些文件随

机地分属于不同的目录。与此相适应,我们就需要在其中某个节点上维持一个集中的目录,这个目录告诉我们什么文件在哪个节点上,然后就各自前去访问。这就像电话的查号服务,或者互联网中的 DNS,所以我们就称之为查名服务。这样,一个或数个节点成为瓶颈的概率下降了,大家都来访问某个目录,未必都是访问同一个文件。至于查名服务,虽然集中于一点,但涉及的计算量和网络流量都很小,不致成为瓶颈。

但是,考虑到大数据处理和 MapReduce 的需要,这样的方式还是不够好。设想我们要计算《红楼梦》的词频。这本书有一百二十回,假定我们就用 120 个 Mapper,每个 Mapper 负责处理一回。但是整本《红楼梦》作为一个文件存放在某一个节点上,这 120 个 Mapper 就都要上同一个节点从这个文件中读取自己所要处理的那一回,这就又挤到一起去了。

这里有个策略上的问题,即应该让数据跟着计算走,让数据跑到计算那儿去?还是让计算跟着数据走,让计算跑到数据那儿去?传统上,当数据量很小的时候,我们都是让数据跟着计算走。文件服务器的使用就是一个典型:计算是在“客户机”上进行的,需要数据就从文件服务器读入。在数据量比较大的时候,网络带宽成了瓶颈,就用带宽更高的 SAN。可是,当数据量进一步增大,到了“大数据”的时候,再让数据跟着计算走就不合适了。这时候,让计算跟着数据走,或者说“数据在哪里,就到哪里去计算”,就比较合适了。

但是因为这一百二十回的内容都在同一个节点上,就把 120 个 Mapper 也集中到一个节点上去计算显然是不合理的,那还不如只用一个 Mapper 哩。怎么办?最好是在存储的时候就把文件拆散,把一百二十回的《红楼梦》分开存储在 120 个节点上,每个节点一回。这样,我们就可以按“数据在哪里,计算就在哪里”的原则配上 120 个 Mapper,每个都是就地读取数据,这样效率就高了。或者,即使某个 Mapper 因为种种原因而放不到目标数据所在的节点上,也可以把它放在尽量靠近的节点上。

当然,《红楼梦》只是很小的一个文件,传输这么一个文件对于现代局域网的带宽而言根本不在话下,这里只是用来说明问题。大数据处理所用文件的大小比之往往要大上千倍、万倍甚至更高。

HDFS,作为一个分布式的文件系统,就是按这样的思路设计的。HDFS 的分布,是“块”这个层次的分布,比上面所说文件这一层次的分布粒度更细。

我们知道,在实际存储的时候,文件是分块的。从逻辑上说,早期的文件里,特别是数据库文件里,都是一个个的“记录”,所以有“记录块”之说。从物理上说,则磁盘上分成一个个的“扇区”,一个扇区就是一“块”。但是记录块和扇区块都很小,扇区块略大一点,也只有 2KB。对于大数据以及现代网络技术来说,这样的量级当然是太小了。所以,HDFS 把文件分成块,这个“块”是虚拟的,其大小可以自由定义,但是最小也得是 1MB,默认值是 64MB,128MB 也很常用。

在 HDFS 中,每个节点承载着若干(可能是几万、几十万)个块的存储,但是这些块随机地分属于不同的文件,当然也分属于不同的目录,而集中的目录和查名服务,则不是告诉你文件在哪里,而是告诉你具体的块在哪里,然后你就自己去访问。HDFS 的查名服务都集中在一个节点上,称为 NameNode,担负文件内容存储的节点则称为 DataNode。

不过这并不意味着 Hadoop 会重新对磁盘进行硬格式化,更不是将 DataNode 上的磁盘格式化成每个扇区就是 64MB,事实上这也不现实。其实 Hadoop 只是依赖其宿主机的文件系统实现信息的存储。在 DataNode 上,不管是 1MB 还是 64MB,一个(HDFS 的)块对于宿主机

而言就是一个文件,其块号就编码在文件名中。从一个 DataNode 读出一个块,实际上是读出一个文件。而在 NameNode 上,则存储的是 HDFS“文件系统”,实际上只是其整个目录树,或称“namespace”的映像。可想而知,这个映像也是作为宿主系统的文件而存储的。

解决了怎么分布的问题,随之而来的是容错的问题。

人们首先会想到的是查名服务。既然整个目录都集中在一点,那要是这个节点出了问题,整个集群岂不就瘫痪了?当然这是个问题,应该为查名服务配上“热备份”,让它时刻准备着,一旦有问题就顶上去。

但是这还不够,每一个块所在的节点都有可能出问题,这意味着每一个块都可能出问题。假定一个节点出问题的概率是 p ,而整个文件都存储在这个节点上,那么这个文件出问题的概率也就是 p 。但是,如果这个文件分布在 100 个节点上,那么任何一个节点出问题都会使这个文件变得不完整,这个文件出问题的概率就大了 100 倍。所以,不光是查名服务要有备份,每个块都得有备份才行。

正是因为这样,HDFS 采取了“狡兔三窟”的策略,每个块都是一式几份(没有主次之分),分别存放在不同的节点上。至于具体是几份则可以配置,默认是三份。这样的三个备份,在 HDFS 中称为一个“复份(replica)”,就是某个逻辑“块”的一个物理副本。相应地,查名服务要提供的就不只是一个块在什么地方,而是这个块的几个复份分别在什么地方。这样,你就可以自己决定从哪个节点读取这个块的复份,万一失败就换一个节点再去读取另一个复份。

HDFS 的分布和容错这两大特点,就是按上述的思路实现的。

有关文件系统的组织(目录)、文件的构成、块、复份及其存储地点的信息当然也是“数据”,但是却有别于代表着文件内容的数据,是比文件内容更高层次的数据,是用于数据管理的数据,所以称为“元数据(meta-data)”。

如前所述,Hadoop 集群中有一个节点是整个文件系统的核心,称为“查名节点”,即 NameNode,或者干脆就称“名节点”,从其作用上说就是文件系统的“主节点”。一般而言,存放在这个节点上的数据都是用于文件系统组织与管理的“元数据(meta-data)”,例如文件系统的目录结构、具体文件的种种属性、文件中的每个逻辑块各有多少个复份、存储在什么节点上等。用户要访问一个文件时,先得上 NameNode,查一下所要读取内容所在的块号,以及该块的各个复份即 replica 所在的节点,然后就直接上其中的某个节点去访问。如果是写访问,那也要由 NameNode 给分配指定一个块号,并给定几个用以存储复份的节点。注意,Hadoop 文件的写操作仅限于添加和删除,而不支持修改(这是指文件内容即数据本身而言,并非指文件系统(目录)映像中的元数据)。

集群中的其余节点用来存储一般数据、即文件内容,所以称为“数据节点”、即 DataNode。通常一个 Hadoop 集群中会有很多 DataNode。

为 NameNode 提供热备的节点,则称为“Standby NameNode”。或者,也可以分别称为主、次查名节点,即 ActiveNN 和 StandbyNN,这里的“主”是处于活跃状态正在“当班”的意思。

当然,ActiveNN 和 StandbyNN 之间得有同步才能保持一致。然而如果每次有一点改变时就得同步一下,例如每次增加一个子目录、每次创建或删除一个文件、每次改变文件的属性、每次往文件中增添一个块,都得要在主、备两个 NameNode 之间同步一下,那也不胜其烦,系统的开销太大。所以 HDFS 采用了一个变通的办法。

这个办法是这样的:双方从相同的文件系统映像开始,各自保存着一个映像副本,每当有

文件系统变动的要求时,当班的 ActiveNN 对其映像中的元数据作出相应的变动,并将这变动记录在一个日志(Log)文件中,称为 EditLog,但是并不通知 StandbyNN,更不用等待其完成操作。这个 EditLog 可以(但不是必须)既不是在 NameNode 所在的节点上,也不是在 Secondary NameNode 所在的节点上,而是在某个双方都能访问的第三方那里,而且一般要求这第三方具有更高的可靠性。比方说,如果集群中的机器都是普通的商品 PC 机,则 EditLog 就应该放在更可靠、质量更好的服务器上。这样,如果 ActiveNN 出了问题,StandbyNN 出来接管,这时它手头的文件系统映像很可能还是老的,已经远远落后于形势。而手头有着文件系统最新映像的原 NameNode,则此时已经访问不到了。怎么办呢?这时候新的 NameNode,即原先的 StandbyNN,就在其老的、落后于形势的那个映像上用 EditLog 中的记录逐条“重演”一遍。当然,这并不意味着采取所有实际的文件操作(因为存储在诸多 DataNode 上的那些块并未受到影响),而只是针对其文件系统映像中的元数据重演一遍所做的修改。这样,完成了重演之后,这个新 NameNode 上的文件系统映像就赶上了形势,与原 ActiveNN 上的映像一致而得到了同步。可想而知,这样的同步也不一定非要到 NameNode 出问题的时候才做,StandbyNN 完全可以每过一段时间就做一下。像这样双方同步的文件系统映像,或者说两个文件系统映像之间的同步点,就是 checkpoint。一旦双方的文件系统映像得到同步,EditLog 中的记录就被作废,于是一个新的过程又开始了。显然,这跟系统中只有一个 NameNode 而遭遇中途断电重启时的情景是一致的,所以 checkpoint 在只有一个 NameNode 的情况下也同样需要。要是 NameNode 不出问题,在老一些的版本中,这要到下一次重启系统时才会同步。但在新的版本中可以设置过一段时间就同步一下,或者每当 EditLog 达到一定长度就同步一下。

既然 NameNode 把对于文件系统的任何变动都以 EditLog 信息的形式发送出去,则 EditLog 信息的接收者就可以不限于一家,对 EditLog 信息的处置方式也不止一种。其中之一是只作持久存储;另一种是不作持久存储,但在内存中维持一个 NameNode 上文件系统映像的镜像,每有 EditLog 信息到来就对这镜像进行相应的更新修改,这样的节点称为 BackupNode;还有一种方式是这二者的折中,每过一段时间就用累积的 EditLog 信息产生一个新的 Checkpoint。显然,这样可以加快 NameNode 出问题时的接管过程。

所谓 Secondary NameNode,简称 Secondary 或 snn,就是专门用来帮助 NameNode 生成 checkpoint,以减轻 NameNode 负担的。

当然,这些措施使系统变得复杂起来。所以,有容错和没有容错,系统的复杂程度是大不一样的。

在 Hadoop 的代码中,NameNode 是个 Java 类,在一个独立的 JVM 上(作为操作系统的一个进程)运行。同样,SecondaryNameNode、DataNode 也是这样的 Java 类。

Hadoop 提供了一个名为 hdfs 的启动脚本,用来在一个节点上启动这些 Java 类的运行。在 Linux 上的命令行是这样的:

```
hdfs [ -- config confdir ] COMMAND
```

这里的 hdfs 是一个 shell 脚本,--config 是可选项,具体要启动的目标取决于参数 COMMAND。它可以是 namenode、secondarynamenode、datanode,也可以是别的 Java 类名,例如 dfs、dfsadmin、fsck,还可以是 balancer、haadmin、crypto、version 等,甚至还可以是用户所

给定的其他对象名。

当然,在一个集群中,我们在指定用作 NameNode 和 Secondary 的节点上分别启动 NameNode 和 SecondaryNameNode,其余节点上则启动 DataNode。

11.2 目录节点 NameNode

如前所述,HDFS 的结构是一种主/从结构,负责提供查名服务的 NameNode 扮演着主节点的角色。我们不妨大致浏览一下 NameNode 这个类的摘要,先看其结构部分。

```
class NameNode implements NameNodeStatusMXBean{
    ] FSNamesystem namesystem          //HDFS 的目录系统
    ]] FSDirectory dir                  //目录框架
    ]] BlockManager blockManager       //块管理器
    ]] FSImage fsImage                 //需要保存在磁盘上的文件系统映像
    ] NameNodeHttpServer httpServer    //为通过 HTTP 浏览器管理 HDFS 提供 Web 服务
    ] NamenodeRegistration nodeRegistration //这个 NameNode 的登记信息
    ] NameNodeRpcServer rpcServer      //提供 RPC 服务
    ] JvmPauseMonitor pauseMonitor     //监视 JVM 的运行是否停滞
    ] String clientNamenodeAddress     //Namenode 的网址
```

NameNode 上最大、最重要的成分,或者说部门,是查名系统 FSNamesystem,那就是 HDFS 的整个目录系统。如果你觉得 NameNode 的结构看似并不复杂,那是因为 FSNamesystem 很大、很复杂,把 HDFS 的复杂性都集中到那里了,这在后面会另辟一节专门介绍。

至于 NameNodeHttpServer,顾名思义,是一个 HTTP 的 Server,使用户可以用浏览器通过 HTTP 连到 NameNode 上,用来启动文件操作或观看运行状态和种种统计信息。

NameNodeRpcServer 显然就是为客户端提供 RPC 服务的 Server,同类的 Server 我们在前述 YARN 框架中早已见过。

这里的 JvmPauseMonitor,看其类名可知是用来监视 JVM 的运行是否停滞的。其实原理挺简单,这是一个线程,这个线程基本上老是在睡眠,只是周期性地醒来一下,一醒来就从系统获取当时的实际时间,看刚才实际睡眠的时间是否比自己指定要睡眠的时间长出太多。如果长出太多的话,就说明本线程被调度运行的机会已大大减少,系统已经太忙,因而就在运行日志中发出一条警告信息。

至于字符串 clientNamenodeAddress,则只是 Namenode 的网址,即 URL。这是供 client 即客户端使用的。

再看 NameNode 操作部分的摘要。

```
class NameNode implements NameNodeStatusMXBean{
    ] NameNode()          //构造函数
    > setClientNamenodeAddress(conf)
    > namenodeId = HAUtil.getNameNodeId(conf, nsId)
```



```

> this.haEnabled = HAUtil.isHAEnabled(conf, nsId)
> state = createHARState(getStartupOption(conf))
> this.haContext = createHAContext()
> initialize(conf)
> state.enterState(haContext)
] format(Configuration conf)
] initialize(Configuration conf)
] doRollback(Configuration conf, ...)
] doRecovery(...)
] createNameNode(String argv[], Configuration conf) //根据命令行选项做不同的事
> startOpt = parseArguments(argv)
> switch (startOpt) { //启动 NameNode 时的命令行选项
> case FORMAT: //HDFS 的格式化(注意,不同于一般文件系统的格式化)
>+ format(conf, startOpt.getForceFormat(), ...)
>+ terminate() //完成格式化后就退出了
> ...
> case ROLLBACK: //回滚到原先的版本
>+ doRollback(conf, true)
>+ terminate() //做完 Rollback 就退出了
> case BOOTSTRAPSTANDBY:
//使安排用于 NameNode 热备份的节点复制当前 NameNode 的内容,以形成热备份
>+ BootstrapStandby.run(toolArgs, conf) //用于 HA,我们暂时还不关心这个
>+ terminate()
> case BACKUP: case CHECKPOINT: //启动本节点为 backup 或 checkpoint 节点
>+ return new BackupNode(conf, role) //这两种节点的作用见前述
> case RECOVER:
>+ NameNode.doRecovery(startOpt, conf) //试图修复损坏了的 HDFS 文件系统目录
>+ return null //修复完就退出了
> case UPGRADEONLY: //仅进行版本更新升级
>+ new NameNode(conf) //创建 NameNode 过程中会进行版本更新升级
>+ terminate(0) //创建之后就退出运行
> default: //没有使用任何选项,才是真正要启动 NameNode
>> new NameNode(conf)
> }
] main(String argv[])
> namenode = createNameNode(argv, null)

```

我们从 main() 开始,这儿就是对 createNameNode() 的调用。到了 createNameNode() 中,就要根据命令行选项分情形处理了。

比方说选项 format,就是这里的“case FORMAT”,就只是 HDFS 文件系统的格式化,实际上就是其目录系统的格式化,那就是在宿主文件系统中创建一套目录,用来存放 HDFS 的

元数据文件。而且,一旦完成格式化,程序就 `terminate()`, 结束了,因为这个命令行的意图其实并非启动 `NameNode`,而仅仅就是格式化。当然,在未经格式化之前,`NameNode` 是无法正常运行的。

那么为什么格式化要跟 `NameNode` 放在一起呢? 这是因为 HDFS 的格式化只需在 `Namenode` 上进行,也只应在 `Namenode` 上进行。这里所谓的“格式化”,实际上只是在本地文件系统,即宿主机的文件系统中创建几个目录,以及代表着 HDFS 目录结构的映像文件。这个过程与磁盘的格式化毫无关系,跟单机上的文件系统格式化也是两码事。

读者也许疑惑:文件的内容不是要以块为单位存储在数据节点上吗,那为什么数据节点就不需要格式化呢? 前面讲过,HDFS 的数据“块”,其实是以文件的形式存储在数据节点上的宿主文件系统中。而且这里还有个重要的不同。

宿主文件系统中的块号是相对块号,如果有两个磁盘,那么两个磁盘是独立编块号的,而且一个块的块号就反映了它在磁盘上的物理位置,系统必须知道每个具体的磁盘有多少个块,哪些块已经用掉了,哪些还是空闲的。对处于底层的宿主文件系统而言,这样做实属必须。但是 HDFS 就不同了,它是全局统一编块号的,用的是绝对块号,而且块号与物理位置之间也没有固定的联系。

HDFS 的一个块,对于数据节点的宿主文件系统而言就是一个固定长度的文件,块号就是它的文件名。之所以 HDFS 可以这样,就是因为它是建立在宿主文件系统的基础上。

这样,`NameNode` 实际上并不需要知道具体的块在哪个数据节点的什么位置上,而只要知道哪些节点上存储着哪些块,以及节点上还有多少容量可供使用,就行了。进一步,什么块在什么节点上,这个信息其实不必固定记载在 `NameNode` 的文件系统映像中,而是可以在运行的时候让数据节点自行报告。在集群的环境中,`NameNode` 对于数据节点的控制不像单机对于磁盘那样严密,加之每个块都有多个副本保存在不同的节点上,这样比把这些信息固定记载在 `NameNode` 的文件系统映像中更合适,因为保不住哪个节点突然就离线了,过一会儿又回来了,甚至保不住哪个节点上的磁盘被拆装到了另一个节点上。

除 `format` 以外,还有 `rollback`、`recover` 等选项也类似,都不是为启动 `NameNode`,而只是为了对 HDFS 文件系统执行某种操作。这里 `rollback` 是要让文件系统的软件退回到先前的版本,`recover` 则是要修复已经损坏了的 HDFS 文件系统,恢复到上一个 `Checkpoint` 的状态。

只有在不用那些选项中的任何一项的时候,才是真正要启动运行 `NameNode`,所以上面的 `swith` 语句中是在 `default` 下面才创建 `NameNode` 对象。

创建 `NameNode` 对象,就要执行其构造函数,那当然就是 `NameNode()`。从摘要可见其中最重要的操作就是 `initialize()`,其余都与 HA,即“高可用”容错有关,属于一些预备性的操作。完成了 `NameNode` 的初始化之后,就再没有什么主动的操作了,因为 `NameNode` 本质上就是一个 `Server`。

我们看一下 `NameNode.initialize()` 的摘要:

```
NameNode.initialize(Configuration conf)
> loginAsNameNodeUser(conf)
> startHttpServer(conf)    //创建并启动 NameNodeHttpServer
>> httpServer = new NameNodeHttpServer(...)
>> httpServer.start()
```

```

> loadNamesystem(conf) //装载 FSNamesystem
>> FSNamesystem.loadFromDisk(conf) //目前只有从磁盘装载这么一个途径
>>> fsImage = new FSImage(conf, ...)
>>> namesystem = new FSNamesystem(conf, fsImage, false)
>>> namesystem.loadFSImage(startOpt) //真正装载的就是文件目录的映像
>>> nnMetrics = NameNode.getNameNodeMetrics() //用于统计信息
> rpcServer = createRpcServer(conf)
>> new NameNodeRpcServer(conf, this)
> pauseMonitor = new JvmPauseMonitor(conf)
> pauseMonitor.start()
> startCommonServices(conf)
>> namesystem.startCommonServices(conf, haContext) //见下面的 FSNamesystem 摘要
    == FSNamesystem.startCommonServices(conf, haContext)
>> registerNNSMXBean() //用于通过 JMX 对 NN 状态的监视,我们不关心
>> httpServer.setFSImage(getFSImage()) //让 NameNodeHttpServer 知道 FSImage
>> rpcServer.start() //启用 RPC 服务
>> plugins = conf.getInstances(DFS_NAMENODE_PLUGINS_KEY, ServicePlugin.class)
    //从配置文件读取“dfs.namenode.plugins”
>> for (ServicePlugin p: plugins) p.start(this) //如果有的话,就启用所有的 ServicePlugin
>> LOG.info(getRole() + " RPC up at: " + rpcServer.getRpcAddress())

```

显然, NameNode 的初始化主要就是查名系统即 FSNamesystem 的装载,但是 FSNamesystem 是相当大的一个部门,并非完全都要从磁盘加载,需要加载的只是文件系统的元数据映像 FSImage。

NameNode 的作用,简单地讲,是给定一个文件路径名,就告诉你这个文件有哪些块,以及每个块有几个复份,分别存储在什么节点上。这里面有两个环节。第一个是从文件路径名到一个“块表”的映射。这就得有一个类似于 Linux 上那样的目录,然后跟随文件路径名的指引爬目录树,最后得到代表着目标文件的 INode,即“索引节点”,那实际上是一个表,或者说一个清单,里面的每个表项都是一个块号。本来,如果是在单机上,这个块号可以就是文件所在磁盘上的块号,因为整个文件就在同一个磁盘上。但是 HDFS 不一样,它的文件是跨节点分布的,而且每个块都有好几个复份。于是必须有第二个环节,这又是一个表,对于使用中的每一个块号,这个表可以告诉你它有几个复份(replica),分别存储在什么节点上。不过它不用告诉我们具体的复份对应于当地的第几个块或者扇区,因为复份都是作为文件存储的,块号就是文件名。也就是说,这最后一个环节是由当地的宿主文件系统提供的。

这两个环节,即从一个文件的路径名到它的块表,再从具体块号到其复份所在数据节点的两层映射,性质上是有区别的。其中的第一个环节,说明每个具体的文件包含了哪些块,即从文件路径名到块表的映射,就是 HDFS 的目录系统,也跟单机上的目录系统相似,是必须持久存储的,要不然断电之后就什么也没有了。集群中只有 NameNode 才有这方面的信息, DataNode 根本就不知道还有 HDFS 这一层的目录和文件。但是第二个环节,即从块号到存储地点的映射,就不一定了。当然也可以把这个映像持久存储在 NameNode 的文件系统中,

但是这样也有问题,因为 NameNode 对于数据节点其实并没有很强的控制。比方说,可能 NameNode 的记载表明某个数据节点上有某个块号的一个复份,但是那个复份实际上却因为某种原因而不存在了,例如因本地的某种不当操作被删掉了,于是两边就不一致了。再比方说,NameNode 可能记载着某个复份所在处的节点名和 IP 地址,但在实际使用中个别节点的名称和 IP 地址却可能变了。所以,把第二层映射持久存储下来可能是靠不住的。那么怎么办呢? HDFS 所采用的办法是干脆就不持久存储,而让各数据节点定期向 NameNode 报告:我这里有你的这么一些块,你可以让人家需要的时候上我这儿来读取;而 NameNode 则在运行时动态汇集和维持第二层映射所需的信息。

所以,所谓 loadNamesystem(),真正装载的只是用来实现第一层映射的那一部分,就是 HDFS 目录系统的映像,那就是 FSImage。

装载了 FSNamesystem,实际上是 FSImage 之后,HDFS 的文件目录到位了,但是上述第二层映射所需的信息还有待于来自各数据节点的报告。集群中的数据节点会定时向 NameNode 发送心跳(Heartbeat)报告,上面就搭载着这样的信息。

所以,NameNode 至少要经历一轮的 Heartbeat 之后才能真正完成其初始化。

完成了初始化之后,NameNode 就时刻准备着了。

11.3 FSNamesystem

NameNode 的核心是 FSNamesystem,这就是 HDFS 的目录系统。同时,从某种意义上说,FSNamesystem 也是关于集群内众多 DataNode 的记账系统。如果把那些 DataNode 看成一个一个的仓库,那么 FSNamesystem 就是集中的账务管理部门。

我们看 FSNamesystem 的摘要,也是先看其结构部分:

```
class FSNamesystem implements Namesystem, ... {}
] FSDirectory dir //目录框架
] BlockManager blockManager //块管理器
] DatanodeManager datanodeManager //管理着与诸多 DataNode 相关的信息
] HeartbeatManager heartbeatManager //管理着与诸多 DataNode 的联络
] CacheManager cacheManager //缓存管理器
] GSet<CachedBlock, CachedBlock> cachedBlocks //记录着缓冲在所有 DataNode 上的块
] DatanodeStatistics datanodeStatistics //数据节点统计信息
] String blockPoolId
] NameNodeResourceChecker nnResourceChecker //为资源检查线程提供操作方法
] SequentialBlockIdGenerator blockIdGenerator //块号生成器
] NNConf nnConf //NameNode 的配置信息
] FSImage fsImage //需要保存在磁盘上的文件系统映像
```

HDFS 文件系统的目录映像需要持久存储,开机时则需要从磁盘装载文件系统映像,实际上就是目录映像,在一定的条件下则又需要把发生变化后的映像存回磁盘。所以,代表着目录映像的 FSImage 当然是 FSNamesystem 的一个重要部件。不过,从外部看来 FSImage 就是一个二进制映像,而看不到其内部的目录结构,所以还必须有个能将其体现为目录,从而提供实现

前述第一层映射的成分,这就是 `FSDirectory`。有了 `FSDirectory`,我们就可以根据一个文件的路径名查得其块表。

但是如前所述,我们还需要有第二层的映射,这就是 `BlockManager` 的作用。这个对象管理着每个块的各个复份(replica)在集群中各个数据节点上的存放。另外,HDFS 文件系统的块号是统一管理的,与数据节点无关,所以还需要有个顺序生成统一块号的 `SequentialBlockIdGenerator` 对象。显然,这些部件都是必不可少的。

文件内容既然分块存储在各个数据节点的磁盘上,读的时候就得从数据节点的磁盘读入到内存,才能将其发送给客户。但是考虑到既然读到了内存,就不宜一用即弃,而不妨在本地的内存容量允许的条件下缓存(Cache)一段时间,因为说不定过一会儿别的客户也要来读这同一块数据。而且这个情况应该让 `NameNode` 知道,以后有客户需要读取同一块数据时就可以优先指引客户去这个数据节点,这样可以提高效率。所以在 `NameNode` 上,更确切地说是在 `FSNamesystem` 内部,有个 `CacheManager`,它接收来自数据节点关于缓冲情况的报告,维持着一个关于缓冲块的清单。但是当然,一个块被缓冲了并不意味着永远被缓冲,因为数据节点的内存毕竟有限。数据节点会根据 LRU 算法加以调度,长期无人访问的缓冲块就会被丢弃。所以数据节点每隔一段时间就要报告一下,这也是通过 `Heartbeat` 实现的。

另外,这里还有个字符串 `blockPoolId`,是“Block Pool”的 ID。原先一个 Hadoop 集群中只能有一个 `NameNode`,一个 `FSNamesystem`,那就没有“Block Pool”这一说了。可是新版的 Hadoop 开始想要支持多个 `NameNode`,让多个 `FSNamesystem` 共存于一个 Hadoop 集群,它们的数据块则可以杂处共存于数据节点上,称为“联邦(Federation)”模式。于是,就得引进“块池(Block Pool)”的概念,让不同的 `NameNode`,从而不同的 `FSNamesystem`,有属于自己的块池。而“块文件”的文件路径中也得带有相应的信息,带上 `blockPoolId`,以区分不同的 `FSNamesystem`,标明一个块属于哪一个 `FSNamesystem`。当然,如果只有一个 `FSNamesystem`,那么这个 `blockPoolId` 就是空白。

至于 `NameNodeResourceChecker`,则是一个用于资源检查的对象。`FSNamesystem` 内部有个线程 `NameNodeResourceMonitor`,其作用是监视 `NameNode` 上的资源(磁盘容量)使用情况,这个线程周期地调用由 `NameNodeResourceChecker` 提供的操作方法。

对 `FSNamesystem` 的结构有所了解之后,我们再看它操作方法的摘要。

```
class FSNamesystem implements Namesystem, ... {}
] FSNamesystem(Configuration conf, FSImage fsImage, boolean ignoreRetryCache)
    > this.codec = CryptoCodec.getInstance(conf) //映像的存储可能需要加密或压缩
    > this.fsImage = fsImage //作为参数传下来的一个可能有待加载内容的 FSImage 对象
    > this.blockManager = new BlockManager(this, this, conf) //创建块管理器
    > this.datanodeStatistics = blockManager.getDatanodeManager().getDatanodeStatistics()
                                                                    //统计信息
    > this.blockIdGenerator = new SequentialBlockIdGenerator(this.blockManager)
                                                                    //用于生成统一的块号
    > this.fsOwner = UserGroupInformation.getCurrentUser()
    > nameserviceId = DFSUtil.getNamenodeNameServiceId(conf)
    > this.serverDefaults = new FsServerDefaults( ... )
```

```

> this.minBlockSize =      /* 从配置文件获取最小块容量,默认 64MB */
                        conf.getLong(DFSConfigKeys.DFS_NAMENODE_MIN_BLOCK_SIZE_KEY, ...)
> this.maxBlocksPerFile = /* 从配置文件获取单个文件的最大块数 */
                        conf.getLong(DFSConfigKeys.DFS_NAMENODE_MAX_BLOCKS_PER_FILE_KEY, ...)
> this.supportAppends = conf.getBoolean(DFS_SUPPORT_APPEND_KEY, ...)
                        //从配置文件读取是否允许对文件进行 append 操作
> this.dir = new FSDirectory(this, conf)                //创建目录框架
> this.cacheManager = new CacheManager(this, conf, blockManager) //创建缓存管理器
> this.nnConf = new NNConf(conf) //NameNode 的配置信息
] loadFromDisk(Configuration conf) //从磁盘装载文件系统映像,见前节说明
] loadFSImage(StartupOption startOpt) //装载 FSImage,由 loadFromDisk()调用
> if (startOpt == StartupOption.FORMAT) { // startOpt 是命令行中的启动选项
>+ fsImage.format()                //如果要求格式化,就先行格式化
>+ startOpt = StartupOption.REGULAR //然后再按常规处理
> }
> recovery = startOpt.createRecoveryContext() //根据 startOpt 创建用于 Recovery 的 Context
> boolean staleImage = fsImage.recoverTransitionRead(startOpt, this, recovery)
                        //从宿主文件系统读入映像文件和 EditLog,加以合并处理
> if (needToSave) fsImage.saveNamespace(this) //如果需要的话就将合并后的映像写回
> fsImage.openEditLogForWrite() //老的 EditLog 已不再需要,准备写新的 EditLog
> imageLoadComplete()
] startCommonServices(Configuration conf, HAContext haContext)
> nnResourceChecker = new NameNodeResourceChecker(conf) //供资源检查线程使用
> blockManager.activate(conf) //启用 BlockManager
> registerMXBean() //用于通过 JMX 对 NN 状态的监视,我们不关心
> DefaultMetricsSystem.instance().register(this) //用于统计
> snapshotManager.registerMXBean() //此处暂不关心快照(snapshot)机制
] startStandbyServices(conf) //用于作为 NameNode 热备份的节点
] metaSave(String filename) //Dump all metadata into specified file
> file = new File(System.getProperty("hadoop.log.dir"), filename) //创建一个文件
> os = new FileOutputStream(file) //按写操作打开此文件,为其创建一个输出流
> osw = new OutputStreamWriter(os) //以此为基础创建一个 OutputStreamWriter
> b = new BufferedWriter(os) //再以 OutputStreamWriter 为基础创建 BufferedWriter
> out = new PrintWriter(b) //再以此为基础创建 PrintWriter
> metaSave(out) //将 PrintWriter 作为参数传给 metaSave(PrintWriter out),Dump 元数据
>> blockManager.metaSave(out) //调用 BlockManager.metaSave()
> out.flush()
] registerDatanode() //接受 DataNode 登记
] handleHeartbeat() //接受并处理 DataNode 的心跳信号
] class NameNodeResourceMonitor implements Runnable {} //资源检查线程

```



```

] class NameNodeEditLogRoller implements Runnable {} //EditLog 跟进线程
] datanodeReport() //为客户提供 DataNode 报告
] saveNamespace() //保存当前的 Namespace,即文件系统映像 FSImage
    > checkOperation(OperationCategory.UNCHECKED) //与热备容错有关,暂不关心
    > checkSuperuserPrivilege() //需要 Superuser 权限
    > cacheEntry = RetryCache.waitForCompletion(retryCache)
        //将映像保存在宿主文件系统的 fsimage 文件中,并创建空白 edits 文件
    > getFSImage().saveNamespace(this) //调用文件系统映像这一层的保存操作
    >> imageTxId = getLastAppliedOrWrittenTxId()
    >> saveFSImageInAllDirs(source, nnf, imageTxId, canceler) //文件系统映像涉及多个目录
    >> storage.writeAll() //写所有存储文件。Storage 对象代表着有关存储信息的文件
    >> storage.writeTransactionIdFileToStorage(imageTxId + 1) //写 TransactionId 文件

```

摘要中已加注释,限于篇幅此处就不详述了,读者可按此提示进一步阅读源码。

其中有些函数,如 registerDatanode()、handleHeartbeat()、startStandbyServices(),还有 loadFSImage(),后面还要结合流程加以讲解。

11.4 文件系统目录 FSDirectory

我们说 FSNamesystem 是 HDFS 文件系统的核心,是因为它实现了 HDFS 的目录机制,然而这个目录究竟是怎么个东西呢?

目录是个由“索引节点(index node)”构成的树状结构,树上的每个节点都是一个索引节点。索引节点至少有两种:一种是作为一级目录的索引节点,称为目录节点,目录节点可以出现在树上的任何位置上;另一种是作为一个文件的索引节点,称为文件节点,文件节点只能出现在树枝的末梢即被称为“叶节点”的位置上。每个节点都有个节点名,树的根是一个名为 Root 的目录节点,以符号“/”表示。

一个文件节点或目录节点的全路径名,是从根节点开始向该节点顺序推进,直至到达该节点的全路径上所有节点的排列,中间以“/”字符分隔,例如“/dir1/dir12/dir123/file1”。

目录节点与文件节点同为索引节点,但是节点的内容和作用不同。目录节点的主要内容是一个索引节点的表列(List),里面可以有代表着子目录的目录节点,也可以有文件节点。文件节点的主要内容则是一个“块描述结构”数组,这个数组的每个元素都描述和代表着文件中的一个块。

在 HDFS 的代码中,索引节点 INode 是一个抽象类,因为具体的索引节点可以是目录节点或文件节点,或者也可以是符号连接节点或所谓 Reference 节点,INode 只是所有这些节点结构的公共头部。我们先看一下 INode 的摘要:

```

abstract class INode implements INodeAttributes, Diff.Element<byte[]> {}
] INode parent //指向父节点
] isDirectory() //判断本 INode 是否为目录节点
    > return false //暂定不是,除非本 INode 被扩展成目录节点
] INodeDirectory asDirectory() //把本 INode 当作目录节点,暂不允许

```

```

    > IllegalStateException("Current inode is not a directory: " + this.toDetailString())
] isFile()                //判断本 INode 是否为文件节点
    > return false         //暂定不是,除非本 INode 被扩展成文件节点
] INodeFile asFile()       //把本 INode 当作文件节点,暂不允许
    > IllegalStateException("Current inode is not a file: " + this.toDetailString())
] isSymlink()              //判断本 INode 是否为符号连接节点
    > return false         //暂定不是,除非本 INode 被扩展成符号连接节点
] INodeSymlink asSymlink() //把本 INode 当作符号连接节点,暂不允许
    > IllegalStateException("Current inode is not a symlink: " + this.toDetailString())
] isReference()            //判断本 INode 是否为转引节点
    > return false         //暂定不是,除非本 INode 被扩展成转引节点
] INodeReference asReference() //把本 INode 当作转引节点,暂不允许
    > IllegalStateException("Current inode is not a reference: " + this.toDetailString())

```

我们知道,抽象类是不能实体化的,一定得经过扩充变成实体类才能实体化,所以代码中一定会有经过扩充的 INode。在所有扩充了 INode 的类中,INode 的位置一定处于头部,起着这些类的“根”的作用。

从结构上说,INode 只有一个成分,就是 parent,用来说明本 INode 的父节点是谁。别的成分就都有待扩充了。

INode 所提供的操作方法,如 isFile()、asFile()等,则表明 INode 并非任何可以实际存在的索引节点,因为所有这些操作方法不是返回 false 就是引起异常。但是,假定把 INode 扩充为文件节点 INodeFile,则 INodeFile 会以自己提供的 isFile()和 asFile()来覆盖 INode 所提供的同种操作方法,于是 isFile()就会返回 true,asFile()就会把程序中代表着这个 INode 的变量转变成(当成)代表着文件节点的 INodeFile。余可类推。

不过 INode 并非直接就扩充为文件节点 INodeFile 或目录节点 INodeDirectory,而是先扩充为另一个抽象类 INodeWithAdditionalFields:

```

abstract class INodeWithAdditionalFields extends INode implements LinkedElement {}
] long id                //节点号
] byte[] name            //节点名
] long permission        //访问许可
] long modificationTime  //最后一次修改的时间
] long accessTime        //最后一次访问的时间
] LinkedElement next     //链表中的下一个元素
] Feature[] features     //各种属性,例如访问控制名单 Acl 等
] INodeWithAdditionalFields(INode parent, long id, byte[] name, long permission, ...)
] setPermission(FsPermission permission)
] setAccessTime(long accessTime)
] addFeature(Feature f)
] ...

```

之所以要先扩充成这样一个抽象类,是因为这里所添加的种种成分和操作方法是文件节

点 `INodeFile`、目录节点 `INodeDirectory` 和符号连接节点 `INodeSymlink` 所共有的, 相当于是从这三者提取的公因子。那么为什么不把这些成分直接就放在 `INode` 中呢? 这是因为还有一种节点 `INodeReference`, 即“转引节点”, 是不具有这些成分的, 那得直接从 `INode` 扩充得来。注意, “`class INodeWithAdditionalFields extends INode`”说明这是 `INode` 的子类, 也即 `INodeWithAdditionalFields` 在结构上包含了 `INode`, 并且 `INode` 在其数据结构的头部。

`INodeDirectory` 和 `INodeFile` 则是对 `INodeWithAdditionalFields` 的扩充。我们先看 `INodeDirectory` 的摘要:

```
class INodeDirectory extends INodeWithAdditionalFields ...{  
    ] List<INode> children    //当前目录下面的所有子节点  
    ] searchChildren(byte[] name) //在当前目录中寻找名为 name 的节点  
        > children == null? -1: Collections.binarySearch(children, name) //对分搜索  
    ] nextChild(ReadOnlyList<INode> children, byte[] name)  
    ] addChild(INode node)      //在当前目录中增加一个节点, 同名的操作方法有三个  
        > low = searchChildren(node.getLocalNameBytes())  
        > if (low >= 0) return false //同名节点业已存在, 不能再加  
        > addChild(node, low)      //见下面的 addChild(INode node, int insertionPoint)  
    ] addChild(INode node, boolean setModTime, int latestSnapshotId)  
        > low = searchChildren(node.getLocalNameBytes())  
        > if (low >= 0) return false //同名节点业已存在, 不能再加  
        > if (isInLatestSnapshot(latestSnapshotId)) ... //我们暂不关心 Snapshot  
        > addChild(node, low) //见下面的 addChild(INode node, int insertionPoint)  
    ] addChild(INode node, int insertionPoint)  
        > if (children == null){ //如果动态数组 children 尚未创建, 就创建之  
        >+ children = new ArrayList<INode>(DEFAULT_FILES_PER_DIRECTORY)  
            //原始的数组大小默认为 DEFAULT_FILES_PER_DIRECTORY, 即 5  
        > }  
        > node.setParent(this)  
        > children.add(-insertionPoint-1, node) == ArrayList<INode>.add(-insertionPoint-1, node)  
            //将节点 node 加入动态数组 children 中。如果超出数组的现有容量就自动加以扩充  
    ] isDirectory() //这个操作方法覆盖(取代)INode 中的 isDirectory()  
        > return true //返回 true, 因为这确实是目录节点  
    ] INodeDirectory asDirectory() //既然是目录节点, 就可以把 INode 当成 INodeDirectory  
        > return this //作为一个 INodeDirectory 对象
```

当然, 这是摘要, 不是全部, 它还有别的操作方法, 包括其构造函数。

从摘要中可见, 在 `INodeWithAdditionalFields` 的基础上, `INodeDirectory` 所增添的结构成分主要就是一个 `List<INode>`, 这是一个以 `INode` 为元素的动态数组, 用来容纳当前目录中所有直接的子节点。

而操作方法, 除用来覆盖替代的 `isDirectory()` 和 `asDirectory()` 之外, 主要有 `addChild()` 和 `searchChildren()`, 前者把一个 `INode` 节点添加到这个目录中, 后者则在这个目录中寻找给定

节点名的 `INode`, 不管其为文件节点还是目录节点。

实际上还有例如 `removeChild()`, 那跟 `addChild()` 是同一类的操作。还有许多别的操作, 那就不是我们此刻要关心的了。

注意 `INodeDirectory` 结构上包含了 `INodeWithAdditionalFields`, 从而也就包含了 `INode`, 所以 `INodeDirectory` 也提供 `isFile()`, 但那就是 `INode` 中的那个 `isFile()`, 所以会返回 `false`, 表示这不是文件节点。

对于目录节点, 最重要的操作当然是在其下面添加子节点, 即 `addChild()`。这里有三个同名的 `addChild()`, 但是它们的参数表不同, 所以是三个不同的函数。其中真正起作用的是 `addChild(INode node, int insertionPoint)`, 它把子节点 `node` 插入动态数组内指定的位置 `insertionPoint` 上, 如果 `insertionPoint` 为 `-1`, 就放在现有元素的后面。要是超出了数组的容量, 则会自动调整其容量。而 `List<INode>` 的 `add()` 操作则由 Java 库提供。

再看 `INodeFile` 的摘要。

```
class INodeFile extends INodeWithAdditionalFields implements ...{}
] long header //preferredBlockSize, replication 和 storagePolicyID 三种信息编码在一起
] BlockInfo[] blocks //这个数组中的每个元素都是对于一个块的描述
] valueOf(INode inode, String path, boolean acceptNull)
] isFile() //判断是否为文件节点, 用来替代 INode 中的同名函数
  > return true //返回 true, 因为这确实是文件节点
] asFile()
  > return this //返回的是 INodeFile (而不是 INode)
] toUnderConstruction(String clientName, String clientMachine) //把文件设置成正在构建中
] toCompleteFile(long mtime) //把文件设置成已完成构建
] setBlock(int index, BlockInfo blk) //使 blk 成为本文件的第 (index + 1) 个块
  > this.blocks[index] = blk
] setFileReplication(short replication) //设置文件的块存储副本数量
] setStoragePolicyID(byte storagePolicyId)
] getHeaderLong() //读取文件节点的 header
  > return header
] BlockInfo[] getBlocks() //读取本文件的 blocks 数组
  > return this.blocks
] addBlock(BlockInfo newblock)
  > if (this.blocks == null) this.setBlocks(new BlockInfo[]{newblock})
    //如果还没有 blocks 数组就创建之。这发生在文件的第一个块
  > else { //不是第一个块, 已经有了 blocks 数组
    >+ size = this.blocks.length //获取当前的数组大小
    >+ newList = new BlockInfo[size + 1] //新建一个只比原来大 1 个元素的数组
    >+ System.arraycopy(this.blocks, 0, newList, 0, size) //把老数组的内容复制到新数组中
    >+ newList[size] = newblock //把新块添加在新数组的最后
    >+ this.setBlocks(newlist) //把新数组设置成文件的 blocks 数组
```

```
>+> this.blocks = blocks
```

```
> }
```

可见 `InodeFile` 在 `InodeWithAdditionalFields` 的基础上增添的成分主要是一个 `BlockInfo` 数组,文件中包含了多少个块,这个数组就相应有多大。这个数组中的每个元素都是对于一个块的描述。

另一个结构成分 `header` 是把三种信息编码在一起而成的无符号整数。这三种信息是 `preferredBlockSize`、`replication` 和 `storagePolicyID`。其中 `preferredBlockSize` 是文件主所希望的块的大小,默认值为 64MB; `replication` 是副本的个数,默认值为 3; `storagePolicyID` 是块的存储策略编号。

什么叫块的存储策略呢? 我们知道一个块有几个副本,即复份,假定说是三个,这三个副本会被分别存储在不同的数据节点上。可是,怎么存储却是有讲究的,这里面有个策略的问题。最简单而直截了当的策略,是全都直接存储在磁盘上,可是把数据写入磁盘需要时间,如果每次写访问都要等待写入磁盘完成,就等待太久了。那就换个策略,先全都缓存在数据节点的 `RAM_DISK` 中,实际上是内存中,然后从长计议,定期把 `RAM_DISK` 中的这些副本写入磁盘。这样当然是最快的了,但是万一这三个节点都在把缓存着的副本写盘之前就坏了,那么这个块就丢了。所以更好的策略可能是把其中的一个副本直接写入磁盘,另两个可以先缓存在 `RAM_DISK` 中。另外,可以存储的地方也不止磁盘(DISK)和 `RAM_DISK` 这么两个,还可以有固态硬盘 SSD,还可以有别的存档设备 ARCHIVE,它们的读写速度各不相同。这样,仍以三个副本为例,可能有的组合数量,即存储策略数量就不小了。像这样的一条策略,比方说把第一个副本存在 DISK,第二个副本存在 `RAM_DISK`,第三个副本存在 SSD,就是一个 `BlockStoragePolicy` 对象。`FSNamesystem` 内部的 `BlockManager` 有个 `BlockStoragePolicySuite`,就是一个数据块存储策略的数组,而 `storagePolicyID` 则是用于这个数组的下标。

在操作方面,因为 `InodeFile` 用自己的 `isFile()` 替换了 `InodeWithAdditionalFields` 也即 `Inode` 的那个 `isFile()`,所以如果调用 `InodeFile` 的 `isFile()` 就会返回 `true`。但是如果调用它的 `isDirectory()`,则会返回 `false`,因为那就是从 `Inode` 继承下来的 `isDirectory()`,并没有被替换, `InodeFile` 只是替换了它的 `isFile()`。

这里最重要的操作当然是 `addBlock()`,因为显然写文件时总难免要往文件中加块。那么 `InodeFile` 有没有提供类似于 `removeBlock()`、`deleteBlock()` 那样的操作呢? 这倒没有的。要提供这样的操作在逻辑上也许不是很困难,但是 HDFS 并不支持这样的操作。在早期 Hadoop 的 HDFS 中,文件都是一次性写成的,写成之后就再也不改变了。这是因为,作为大数据处理平台的 Hadoop,输入文件的内容只是供反复分析和挖掘之用,没有必要去改变它。如果真有必要加以修改,那就另外创建一个新的文件。稍后则增加了对于文件的 `append` 功能,允许在文件尾部增添,但仍旧不允许在文件的中部随机插入和删除。倒是有个函数 `removeLastBlock()`,但那只是用于写入文件的过程中,一旦文件被关闭,那就是永久的了。

数组 `blocks` 是个 `BlockInfo[]`,其每个元素都是一个 `BlockInfo` 对象,里面有 `Block` 的块号,也有关于各个复份存储在什么节点上的信息。其中块号是要持久存储在文件系统映像中的,而复份所在的位置则并不持久存储,这一点后面还会讲到。

知道了目录的构成,我们就可以考察作为 `FSNamesystem` 骨架的 `FSDirectory` 了。我们在前面看到 `FSNamesystem` 内部有两大部件,一个是 `FSImage`,另一个就是 `FSDirectory`。其

实前者是为后者服务的,只是为了解决其持久存储的问题,FSDirectory 才是运行时的“活”的目录系统。下面是 FSDirectory 类的摘要,先看结构部分:

```
class FSDirectory implements Closeable{
] INodeDirectory rootDir      //根目录,即整个文件系统的根节点
] FSNamesystem namesystem    //说明本目录属于哪一个 FSNamesystem
] INodeMap inodeMap          //本目录中所有 INode 的集合,用于快速寻找 INode
]] GSet<INode, INodeWithAdditionalFields> map
```

这里的 rootDir 表面上只是个 INodeDirectory,实际上却是一个目录树的根节点,代表着从这个节点开始的整个目录树。当然,在文件系统创建之初,目录中还什么也没有的时候,就真的只有这么一个根节点。

当目录树变得很大很复杂的时候,如果不给定路径,要在这个目录树上寻找一个 INode 节点并非易事,有可能要遍历整棵树才能找到。为了加快寻找速度,这里有个 INodeMap 类的对象 inodeMap,我们可以把它看成一个集合,或者一个排好序的数组,把整棵树上的所有节点(其实是对节点的引用)都放入这个集合,这样就可比较快地根据节点号从中找到一个节点。INodeMap 是对 LightweightGSet<INode, INodeWithAdditionalFields> 的包装,所以对 INodeMap 的操作实际上就是对 LightweightGSet 的操作。

再看 FSDirectory 的操作方法摘要:

```
class FSDirectory implements Closeable{
] INodeDirectory createRoot(FSNamesystem namesystem)
] INodeFile addFile (String path, PermissionStatus permissions, short replication,
                    long preferredBlockSize, ...) //将一文件节点加入目录
> newNode = newINodeFile(namesystem.allocateNewInodeId(), permissions,
                        modTime, modTime, replication, preferredBlockSize) //先为其创建 INode
> newNode.toUnderConstruction(clientName, clientMachine) //标明其为正在构建中
> added = addINode(path, newNode) //将此 INode(实际上是 INodeFile)加入目录中
> return newNode
] addINode(String src, INode child) //Add the given child to the namespace.
> byte[][] components = INode.getPathComponents(src) //将路径分解成一个数组
> child.setLocalName(components[components.length-1]) //路径中的最后一节是文件名
> addLastINode(getExistingPathINodes(components), child, true)
] addLastINode(INodesInPath inodesInPath, INode inode, boolean checkQuota)
> pos = inodesInPath.getInodes().length-1
> addChild(inodesInPath, pos, inode, checkQuota)
] addChild(INodesInPath iip, int pos, INode child, boolean checkQuota)
> INode[] inodes = iip.getInodes()
> if (checkQuota) { //检查配额
> + verifyMaxComponentLength(child.getLocalNameBytes(), inodes, pos) //节点名不能太长
> + verifyMaxDirItems(inodes, pos) //一个目录下面的节点数不能太多
> }
```



```

> verifyINodeName(child.getLocalNameBytes())
> INodeDirectory parent = inodes[pos - 1].asDirectory() //父节点一定是目录节点
> added = parent.addChild(child, true, iip.getLatestSnapshotId()) //将子节点加在父节点下面
> iip.setINode(pos - 1, child.getParent())
> AclStorage.copyINodeDefaultAcl(child)
> addToINodeMap(child) //同时也把子节点加入 INode 集合中
] addToINodeMap(INode inode) //把一个新的节点加入 INode 集合 INodeMap 中
> if (inode instanceof INodeWithAdditionalFields) { //必须是 INodeWithAdditionalFields
>+ inodeMap.put(inode) //加入 INodeMap 集合,以便快速查找
>+ if (!inode.isSymlink()) {
>++ XAttrFeature xaf = inode.getXAttrFeature() //INodeWithAdditionalFields 有不少属性
>++ ...
>+ }
> }
] getNode(String src) //给定路径名,从目录树中查找一个文件或子目录的 INode
> iip = getLastINodeInPath(src)
> return iip.getINode(0)
] getNode(long id) //给定节点号,从 INodeMap 中获取 INode
> return inodeMap.get(id)
] BlockInfo addBlock(String path, INodesInPath inodesInPath, Block block,
                      DatanodeStorageInfo[] targets) // Add a block to the file
> INodeFile fileINode = inodesInPath.getLastINode().asFile() //获取具体的文件节点
> Preconditions.checkState(fileINode.isUnderConstruction())
//只有标志为正在构建中的文件才可 addBlock()
> blockInfo = new BlockInfoUnderConstruction(block, fileINode.getFileReplication(), ...)
//这是对于 Block 的扩充,在 Block 上添加“正在构建中”标志等相关成分
> getBlockManager().addBlockCollection(blockInfo, fileINode) //知会 BlockManager
> fileINode.addBlock(blockInfo) == INodeFile.addBlock(blockInfo) //见 INodeFile 的摘要
] removeBlock(String path, INodeFile fileNode, Block block) //只适用于正在构建中的文件
> fileNode.removeLastBlock(block)
> getBlockManager().removeBlockFromMap(block)
] Block[] setReplication(String src, short replication, short[] blockRepls) //设置复份数量
> INode inode = iip.getLastINode()
> INodeFile file = inode.asFile()
> file.setFileReplication(replication, iip.getLatestSnapshotId())
> return file.getBlocks()
] setStoragePolicy(String src, byte policyId)
] setPermission(String src, FsPermission permission)
] setOwner(String src, String username, String groupname)
] delete(String src, BlocksMapUpdateInfo collectedBlocks, ...)

```

//删除一个目录及其下面的所有文件,并回收所有的块

```
> filesRemoved = unprotectedDelete(inodesInPath, collectedBlocks, removedINodes, mtime)
>> INode targetNode = iip.getLastINode()
>> removed = removeLastINode(iip)
>> targetNode.destroyAndCollectBlocks(collectedBlocks, removedINodes)
] getFileInfo(String src, boolean resolveLink, boolean isRawPath, boolean includeStoragePolicy)
] setAcl(String src, List<AclEntry> aclSpec)
```

读者不妨从 `addFile()` 开始,顺着 `addFile()` > `addINode()` > `addLastINode()` > `addChild()` 这个流程,再转入前面的 `INodeDirectory.addChild()`,这样走过一遍以后就可对此过程有所理解了。

这里还有个操作方法 `addBlock()`,对于一个目录怎么会有 `addBlock()`? 原来这是针对目录中的具体文件的,参数 `path` 指明了这个文件,另一参数 `inodesInPath` 则以数组的形式给出了沿着路径所经历的所有目录节点,最后一个文件节点。

摘要中已经加了注释,其余的那些成分和方法,就留给读者自己阅读分析了。

11.5 文件系统映像 FsImage

`FSDirectory` 对象只是存在于 `NameNode` 所在节点机的内存中,尽管这机器不会轻易就关机断电,但这样的存在毕竟只是短暂(ephemeral)的,而不是永久的存在。然而 HDFS 文件中那些文件的内容却不会因此而消失,它们的存在是持续的(persistent)。我们总不能让持续存在的文件内容因为目录信息的消逝而变得不可访问,解决这个问题的唯一办法就是使目录信息也得到持久的存储。事实上,单机上文件系统的结构信息从来都是作为“文件卷”的一部分而被持久存储的。但是在像 Hadoop 这样的分布式文件系统中情况却有些不同。假定我们把整个 `FSDirectory` 对象加以串行化并写入磁盘,然后经历关机/开机的周期以后原封不动地加以恢复,是否就能保证正确的文件访问呢?

前面讲过,文件系统的结构信息实际上分成两个层次:第一个层次是从具体文件的路径(和具体内容在文件中的位置)到文件内容块号的映射,这一层映射是固定不变的,应该持久存储。第二层的映射则是从块号到存储地点的映射,在单机上这一层映射也是不变的,事实上关于存储地点(位置)的信息往往就编码在块号中。但是,在像 Hadoop 这样的集群中,这个映射却并不固定,下次开机时甚至有可能这一台机器上的磁盘已经被拆装到了那一台机器上,或者干脆已经不存在了。即使在连续运行的集群上,这样的事也很有可能发生,一个几千个、上万个节点的集群上难免有机器或磁盘损坏。所以,在像 Hadoop 这样的集群上,对这第二层的映射不应采取持久存储,而要让 `DataNode` 动态向 `NameNode` 报告。

而 HDFS 文件系统的 `FSImage` 就代表着需要加以持久存储的目录结构和第一层映射,所以这个映像并不是简单的 `FSDirectory` 串行化,而必须从 `FSDirectory` 中抽取应该持久存储的信息以形成文件系统的映像。下面是 `FSImage` 类的摘要:

```
class FSImage implements Closeable{
] FSEditLog editLog //文件系统变更日志
] NNStorage storage //代表着存储介质,实际上是宿主文件系统的一系列目录和文件
```

```

] NNStorageRetentionManager archivalManager
] FSImageConfiguration conf, Collection<URI> imageDirs, List<URI> editsDirs) //构造函数
> storage = new NNStorage(conf, imageDirs, editsDirs)
> this.editLog = new FSEditLog(conf, storage, editsDirs)
> archivalManager = new NNStorageRetentionManager(conf, storage, editLog)
] format(FSNamesystem fsn, String clusterId) //在宿主文件系统中创建 HDFS 的框架
] recoverTransitionRead(StartupOption startOpt, FSNamesystem target,
                        MetaRecoveryContext recovery)
] recoverStorageDirs(StartupOption startOpt,
                    Map<StorageDirectory, StorageState> dataDirStates)
] doUpgrade(FSNamesystem target) //软件版本升级
] doRollback(FSNamesystem fsns) //回滚版本升级
] rollEditLog() //RditLog 的滚进
] loadFSImage(FSNamesystem target, StartupOption startOpt, MetaRecoveryContext recovery)
//从磁盘(其实是宿主文件系统)装载文件系统映像
] loadEdits(Iterable<EditLogInputStream> editStreams, FSNamesystem target, ...)
//从磁盘(其实是宿主文件系统)装载变更日志
] saveFSImage(SaveNamespaceContext context, StorageDirectory sd, NameNodeFile dstType)
] class FSImageSaver implements Runnable {}
] saveNamespace(FSNamesystem source, NameNodeFile nnf, Canceler canceler)

```

从这摘要中可以看出,FSImage 对象内部并没有以数据形式出现的映像,而是提供了一些生成和处理映像的方法,真正的映像则以文件的形式存在于宿主文件系统中。当创建一个 FSImage 对象时,从它的构造函数可以看出同时也创建了它的三个部件,其中的两个是关键性的:一个是 NNStorage 类的对象 storage;还有一个是 FSEditLog 类的对象 editLog。创建这两个部件所得结果是磁盘上的目录和文件,而不仅仅是内存中的数据结构。

这里我们需要对 NNStorage 有更深入的了解。NNStorage,显然是“NameNode Storage”的意思,就是 NameNode 上的存储手段、存储介质。事实上,NameNode 是以所在节点上宿主文件系统中的文件目录作为“存储设备”的,所以这首先表现为宿主系统上一系列的目录与文件。NNStorage 这个类是对抽象类 Storage 的扩充,而后者又是对 StorageInfo 类的扩充。我们就从最基本的 StorageInfo 类开始:

```

class StorageInfo {}
] int layoutVersion; // layout version of the storage data
] int namespaceID; // id of the file system
] String clusterID; // id of the cluster
] long cTime; // creation time of the file system state
] NodeType storageType; // Type of the node using this storage

```

一个 StorageInfo 对象就是一项存储信息。里面的内容包括存储格局(layout)的版本号 layoutVersion,这是因为 HDFS 的存储格局有过好几次变化,而且还可能会变,软件的版本升级往往就伴随着 HDFS 存储格局的改变。另一个需要说明的成分是 NodeType,这是个枚举

类型,用来说明节点的性质:

```
enum NodeType {
    NAME_NODE,
    DATA_NODE,
    JOURNAL_NODE; //专门记录运行日志的节点 JournalNode
}
```

这是因为,不仅 NameNode 需要存储手段,DataNode 也要,只是所存储内容的性质不同。NameNode 存储的是结构信息,DataNode 存储的是数据,是文件内容。与节点类型相应,NameNode 上有 NNStorage,DataNode 上有 DataStorage,专门用来记录 Log 信息的 JournalNode 上则有 JNStorage。

抽象类 Storage 是对 StorageInfo 的扩充,里面定义了若干目录名和文件名:

```
abstract class Storage extends StorageInfo {}

] String STORAGE_FILE_LOCK      = "in_use.lock" //用于加锁
] String STORAGE_DIR_CURRENT    = "current"     //当前的文件系统结构
] String STORAGE_DIR_PREVIOUS   = "previous"     //先前的文件系统结构,以备回滚
] String STORAGE_TMP_REMOVED    = "removed.tmp"  //记载着已经删除的文件和目录
] String STORAGE_TMP_PREVIOUS    = "previous.tmp" //记载着先前的文件和目录
] String STORAGE_TMP_FINALIZED   = "finalized.tmp" //记载着已经封存的文件和目录
] String STORAGE_TMP_LAST_CKPT   = "lastcheckpoint.tmp" //最近一个 checkpoint
] String STORAGE_PREVIOUS_CKPT   = "previous.checkpoint" //前一个 checkpoint
] enum StorageState {           //文件状态
    NON_EXISTENT,
    NOT_FORMATTED,
    COMPLETE_UPGRADE,
    RECOVER_UPGRADE,
    COMPLETE_FINALIZE,
    COMPLETE_ROLLBACK,
    RECOVER_ROLLBACK,
    COMPLETE_CHECKPOINT,
    RECOVER_CHECKPOINT,
    NORMAL;
}

] class StorageDirectory implements FormatConfirmable {} //用作存储设备的宿主系统目录
]] File root;           // root directory
]] boolean isShared
]] StorageDirType dirType; // storage dir type
] List<StorageDirectory> storageDirs
] ...
```

所谓一个 Storage,实际上就是具体节点上宿主文件系统中的一个子树,该子树的根目录中有

current、previous 等子目录,也可以有例如 previous.tmp、finalized.tmp、previous.checkpoint 等文件。

而 NNStorage 则是对 Storage 的扩充:

```
class NNStorage extends Storage implements Closeable, StorageErrorReporter {}
] String blockpoolID = ""; // id of the block pool
] long mostRecentCheckpointTxId //TxId of the last transaction that was included in the most
    //recent fsimage file. This does not include any transactions
    //that have since been written to the edit log.
] long mostRecentCheckpointTime = 0
] List<StorageDirectory> removedStorageDirs //list of failed (and thus removed) storages
] enum NameNodeFile {} //文件的类型有 10 种,见后
] enum NameNodeDirType implements StorageDirType {} //目录的类型有 4 种
    UNDEFINED,
    IMAGE, //用于存放 IMAGE 文件
    EDITS, //用于存放 EditLog 文件
    IMAGE_AND_EDITS //既可用于存放 IMAGE 文件,也可用于存放 EditLog 文件
}
] NNStorage(Configuration conf, Collection<URI> imageDirs, Collection<URI> editsDirs)
    > super(NodeType.NAME_NODE)
    > storageDirs = new CopyOnWriteArrayList<StorageDirectory>()
    > setStorageDirectories(imageDirs, Lists.newArrayList(editsDirs),
        FSNamespace.getSharedEditsDirs(conf))
] setStorageDirectories(Collection<URI> fsNameDirs, Collection<URI> fsEditsDirs)
] getStorageDirectory(URI uri)
] getImageDirectories()
    > return getDirectories(NameNodeDirType.IMAGE)
] getEditsDirectories()
    > return getDirectories(NameNodeDirType.EDITS)
] readTransactionIdFile(StorageDirectory sd)
] writeTransactionIdFile(StorageDirectory sd, long txid)
    > File txIdFile = getStorageFile(sd, NameNodeFile.SEEN_TXID)
    > PersistentLongFile.writeFile(txIdFile, txid)
] writeTransactionIdFileToStorage(long txid)
    > for (StorageDirectory sd : storageDirs) writeTransactionIdFile(sd, txid)
] getFsImageNameCheckpoint(long txid)
] getFsImageName(long txid, NameNodeFile nnf)
    //return The first image file with the given txid and image type.
] getFsImage(long txid, EnumSet<NameNodeFile> nnfs)
    > for (Iterator<StorageDirectory> it = dirIterator(NameNodeDirType.IMAGE);
        it.hasNext();) {
    >+ StorageDirectory sd = it.next()
```

```

    >+ for (NameNodeFile nnf : nnfs) {
    >++ File fsImage = getStorageFile(sd, nnf, txid)
    >++ if (FileUtil.canRead(sd.getRoot()) && fsImage.exists()) return fsImage
    >+ }
    > }
    > return null
] getFsImageName(long txid)
    > return getFsImageName(txid, NameNodeFile.IMAGE)
] getImageFileName(long txid)
    > return getNameNodeFileName(NameNodeFile.IMAGE, txid)
    >> return String.format("%s_%019d", nnf.getName(), txid)
] String getCheckpointImageFileName(long txid)
    > return getNameNodeFileName(NameNodeFile.IMAGE_NEW, txid)
] getRollbackImageFileName(long txid)
    > return getNameNodeFileName(NameNodeFile.IMAGE_ROLLBACK, txid)
] ...

```

NNStorage 内部定义了一个枚举类型 NameNodeFile, 这告诉我们 NameNode 上可能有些什么类型的文件, 以及它们的文件名前缀是什么:

```

enum NameNodeFile {
    IMAGE      ("fsimage"),
    TIME       ("fstime"), // from "old" pre-HDFS-1073 format
    SEEN_TXID  ("seen_txid"),
    EDITS      ("edits"),
    IMAGE_NEW  ("fsimage.ckpt"),
    IMAGE_ROLLBACK("fsimage_rollback"),
    EDITS_NEW  ("edits.new"), // from "old" pre-HDFS-1073 format
    EDITS_INPROGRESS ("edits_inprogress"),
    EDITS_TMP  ("edits_tmp"),
    IMAGE_LEGACY_OIV ("fsimage_legacy_oiv"); // For pre-PB format

    private String fileName = null;
    private NameNodeFile(String name) { this.fileName = name; }
    public String getName() { return fileName; }
}

```

这里有 10 种, 其中最重要的是前缀为 fsimage 的 IMAGE 文件、前缀为 edits 的 EDITS 文件, 这二者意义自明。还有前缀为 fsimage.ckpt 的文件, 类型为 IMAGE_NEW。以 IMAGE 文件为例, 要往宿主文件系统中写入 IMAGE 文件时, 就通过 getImageFileName() 获取文件名, 这文件名是前缀 fsimage 后面加上本次操作的 Transaction ID, 即 txid, 共 19 位十进制数字, 中间还有个下横线。同理, 如果是 checkpoint 文件就是 fsimage.ckpt 后面加 txid。

所以,NNStorage 在很大程度上就是个工具。

为了存储 HDFS 的目录框架即元数据,还需要在 NNStorage 中建立一些目录。从枚举类型 NameNodeDirType 的定义可以看出,这些目录分为四种类型,也就是四种用途。其中 UNDEFINED 我们可以忽略,其余三种就都很重要,特别是 IMAGE 和 EDITS。可是具体要建一些什么目录呢?这是可以在配置文件 hdfs-default.xml 和 hdfs-site.xml 中设置的。Hadoop 的代码中定义了一个静态类 DFSSConfigKeys,里面有许多静态字符串,其中就有:

```
DFS_NAMENODE_NAME_DIR_KEY = "dfs.namenode.name.dir"
DFS_NAMENODE_EDITS_DIR_KEY = "dfs.namenode.edits.dir"
DFS_NAMENODE_SHARED_EDITS_DIR_KEY = "dfs.namenode.shared.edits.dir"
DFS_NAMENODE_CHECKPOINT_DIR_KEY = "dfs.namenode.checkpoint.dir"
DFS_NAMENODE_CHECKPOINT_EDITS_DIR_KEY = "dfs.namenode.checkpoint.edits.dir"
DFS_NAMENODE_EDITS_DIR_DEFAULT = file:///tmp/hadoop/dfs/name
```

以“dfs.namenode.name.dir”为例,就可以在 hdfs-default.xml 中查得:

```
<property>
<name>dfs.namenode.name.dir</name>
<value>file://${hadoop.tmp.dir}/dfs/name</value>
<description>Determines where on the local filesystem the DFS name node
    should store the name table(fsimage). If this is a comma-delimited list
    of directories then the name table is replicated in all of the
    directories, for redundancy. </description>
</property>
```

这里只提供了一个,但也可以是逗号分隔的一个表列。

把整个目录的内容保存在磁盘上,实际上是宿主文件系统中,就是一个文件系统映像,也就是一个 FSImage 对象。FSImage 有两个构造函数:一个是 FSImage(Configuration conf);另一个是 FSImage(Configuration conf, Collection<URI> imageDirs, List<URI> editsDirs)。其实前者会调用后者,但是如果已经有了 imageDirs 和 editsDirs 就可以跳过前者。

```
class FSImage implements Closeable {}
] FSImage(Configuration conf)
> imgd = FSNamesystem.getNamespaceDirs(conf)
>> return getStorageDirs(conf, DFS_NAMENODE_NAME_DIR_KEY)
> edtd = FSNamesystem.getNamespaceEditsDirs(conf)
>> return getNamespaceEditsDirs(conf, true)
>>> if (includeShared) { //如果可包括共享目录,就把共享目录也收集进来
>>>+ List<URI> sharedDirs = getSharedEditsDirs(conf)
>>>+ for (URI dir : sharedDirs) editsDirs.add(dir)
>>> }
>>> for (URI dir : getStorageDirs(conf, DFS_NAMENODE_EDITS_DIR_KEY)) {
```

```

>>>+ editsDirs.add(dir) //把配置文件中对 dfs.namenode.edits.dir 的设置收集起来
>>> }
>>> if (editsDirs.isEmpty()) { //如果均无配置,就共享 IMAG 目录
>>>+ return Lists.newArrayList(getNamespaceDirs(conf))
>>> } else { //如有配置就按配置
>>>+ return Lists.newArrayList(editsDirs)
>>> }
> this(conf, imgd, edtd) //补上两个参数以后,调用另一个构造函数
] FSImage(Configuration conf, Collection<URI> imageDirs, List<URI> editsDirs)
> this.conf = conf
> storage = new NNStorage(conf, imageDirs, editsDirs)
> ...

```

由此可见,配置文件中对 `dfs.namenode.name.dir` 一项的设置,就是 `imageDirs`;而对 `dfs.namenode.edits.dir` 和 `dfs.namenode.shared.edits.dir` 这两项的设置加在一起就是 `editsDirs`;如果均无设置就共享 `imageDirs`。

这里还要说明,FSImage 对象并不是到了要保存的时候才创建的,而是 NameNode 一经启动就会创建,并且一直存在。

当我们启动 NameNode 的时候,如果这个 NameNode 是全新的,它就要通过 `format()` 创建 HDFS 文件系统的骨架,这时候就会创建一个 FSImage 对象。这时候从用户的角度看文件系统中还没有内容,但其实也并非空白,因为那里面已经有了一些由系统使用的目录和文件。或者启动 NameNode 时也可以要求通过 `doImportCheckpoint()` 从一个 checkpoint 导入文件系统的內容,这有点类似于整个文件卷的复制;这时候也要创建和保存 FSImage 对象,但这就不是代表着空白文件系统的 FSImage 了。

而如果启动的并非全新的 NameNode,则上次关机之前保存的 FSImage 就在磁盘上,只要从磁盘上装载就行,所以此时会通过 `loadFromDisk()` 把文件系统的结构信息装载进来,这时候创建的 FSImage 对象当然是对上次关机之前所保存的 FSImage 的恢复和继续。不过 FSImage 的保存是有时间间隔的,因而保存着的 FSImage 加上 EditLog 的内容才真实反映关机时文件系统的实际状态。在极端的情况下,自从上次开机保存了一次 FSImage 之后一直到关机都没有再被更新,整个运行期间的所有活动都记录在 EditLog 中。

开机启动 NameNode 时的操作之所以比较复杂,除涉及上次关机来不及反映到 FSImage 中的日志 EditLog 外,还可能存在版本升级或回滚的要求,所以 FSImage 还提供了 `doUpgrade()`、`doRollback()`、`rollEditLog()`、`recoverStorageDirs()` 等操作。这种种因素缠在一起增加了阅读理解代码的难度,所以应该先把保存 FSImage 的过程搞懂,然后再看别的过程。此刻我们最关心的就是这里的 `saveNamespace()` 和 `saveFSImage()` 这两个函数和线程 FSImageSaver,其中 `saveNamespace()` 是总的入口。搞清了从 `saveNamespace()` 开始的过程,我们对 HDFS 和 NameNode 就有了基本的理解。

假定一个 HDFS 文件系统已经从无到有建立起来了,其 FSDirectory 对象里面已经有了不少的文件节点和目录节点,这些文件的内容则已经按块分布到许多 DataNode 上,作为当地宿主系统的文件写入磁盘。这时候来了一个 `saveNamespace` 即保存文件系统的要求。注意,

这个 saveNamespace 所要保存的绝非文件的实际内容,那已经分布存储在许多 DataNode 上,要保存的只是 NameNode 上有关文件系统的结构信息,比方说,某个目录节点下有几个文件,这些文件各有多少块,文件的总长度是什么(最后一块未必写满),最后一次写入是在什么时候,等等。这种 saveNamespace 要求的来源可以是内生的,也可以是外来的。例如,假定上次形成 checkpoint 即保存 FSImage 的时间已经很长,或者 EditLog 已经大到了一定程度,就要形成一个新的 checkpoint,这时候就要调用 saveNamespace(),这就是内生的要求。至于外来的要求,如果系统管理员认为有必要,也可以通过 HDFS 管理工具 DFSAdmin 启动命令行“hdfs dfsadmin -saveNamespace”,于是 DFSAdmin 就会调用其 saveNamespace()方法通过 RPC 请求 NameNode 加以执行:

```
class DFSAdmin extends FsShell {}
] saveNamespace()
> dfs.saveNamespace() == DistributedFileSystem.saveNamespace() //这是 DFSAdmin.dfs
>> dfs.saveNamespace() == DFSClient.saveNamespace() //这是 DistributedFileSystem.dfs
>>> namenode.saveNamespace() == ClientProtocol.saveNamespace() //RPC 调用
//namenode 是 NameNode 的 proxy,它实现了 ClientProtocol 界面
== ClientNamenodeProtocolTranslatorPB.saveNamespace()
//在客户端这一边,ClientNamenodeProtocolTranslatorPB 实现了 ClientProtocol 界面
>>>> rpcProxy.saveNamespace(null, VOID_SAVE_NAMESPACE_REQUEST)
//这以下是 protoBuf 的事
```

注意,DFSAdmin 中的成分 dfs 即 DFSAdmin.dfs 是个 DistributedFileSystem 对象,而 DistributedFileSystem.dfs 则是个 DFSClient 对象。DFSClient 就好像通向 NameNode 的门户,它掌控着对于 NameNode 的 RPC 调用。

这个请求到了服务端,即 NameNode 这一边,则:

```
ClientNamenodeProtocolProtos.saveNamespace(controller, request, done) //由 protoBuf 提供
> impl.saveNamespace(controller, request, done)
>> ClientNamenodeProtocolServerSideTranslatorPB.saveNamespace(
    RpcController controller, SaveNamespaceRequestProto req)
>>> server.saveNamespace() == ClientProtocol.saveNamespace()
== NameNodeRpcServer.saveNamespace()
//在服务端这一边,NameNodeRpcServer 实现了 ClientProtocol 界面
>>>> namesystem.saveNamespace() == FSNamesystem.saveNamespace()
```

我们就从 FSNamesystem.saveNamespace()开始往下看。

```
FSNamesystem.saveNamespace() throws AccessControlException, IOException
> checkSuperuserPrivilege() //只有超级用户可以做这个事
> CacheEntry cacheEntry = RetryCache.waitForCompletion(retryCache)
//如果此前已在做这个事,就等待其结果
> if (cacheEntry != null && cacheEntry.isSuccess()) return // Return previous response
//此前正好也在做这个事,并且成功,就不用再做一遍了
```

```

> getFSImage().saveNamespace(this) == FSImage.saveNamespace(this)
>> FSImage.saveNamespace(source, NameNodeFile.IMAGE, null)
//调用 FSImage.saveNamespace()

> success = true
> RetryCache.setState(cacheEntry, success) //操作成功。因为如果发生异常就到不了这里

摘要中略去了有关异常的 try{}finally{}。实际上,如果在 FSImage.saveNamespace()过程中发生异常,程序就到不了最后那两句。

```

可见,FSNamesystem.saveNamespace()实际上就是 FSImage.saveNamespace()。

```
[FSNamesystem.saveNamespace() > FSImage.saveNamespace()]
```

```

FSImage.saveNamespace(FSNamesystem source, NameNodeFile nnf, Canceler canceler)
> assert editLog!= null : "editLog must be initialized" //哪怕没有内容,EditLog 一定得有
> storage.attemptRestoreRemovedStorage()//是否有以前被撤去的存储设备现在又回来了
> boolean editLogWasOpen = editLog.isSegmentOpen() //看看 EditLog 中的当前段是否还开着
> if (editLogWasOpen) editLog.endCurrentLogSegment(true) //如果开着就把它关掉
> long imageTxId = getLastAppliedOrWrittenTxId() //获取最后的那个 TxID
>> return Math.max(lastAppliedTxId, editLog!= null?editLog.getLastWrittenTxId(): 0)
> saveFSImageInAllDirs(source, nnf, imageTxId, canceler)
> storage.writeAll()
>> this.layoutVersion = getServiceLayoutVersion()
>> for (Iterator<StorageDirectory> it = storageDirs.iterator(); it.hasNext();) {
>>+ writeProperties(it.next())
>> }
> if (editLogWasOpen) {
>+ editLog.startLogSegment(imageTxId + 1, true)
>+ storage.writeTransactionIdFileToStorage(imageTxId + 1)
> }

```

所谓保存文件系统映像,并非只是 Image 文件的事,文件系统的更改日志也是个重要的因素,所以先要检验 editLog 是否在位。另外,editLog 中的记录是分段的,保存映像前要把当前段关闭,完成以后再另开一段。记录在 editLog 中的每个记录都有个“日志(操作)编号”TxId,即 Transaction ID,这里的 imageTxId 就是个 TxId。注意,这个所谓 Transaction 并不是指文件操作本身,并不意味着每次文件操作都是原子性的 Transaction,而是指把文件操作记入日志的操作,那是不可分割的原子操作。所以,TxId 实质上就是一个操作记录号。

最后,HDFS 在宿主文件系统上的 IMAGE 文件目录可能不止一个(取决于配置文件),如果有多个就得在每个 IMAGE 文件目录中都保存一份,所以需要 saveFSImageInAllDirs():

```
[FSNamesystem.saveNamespace() > FSImage.saveNamespace() > saveFSImageInAllDirs()]
```

```
saveFSImageInAllDirs(FSNamesystem source, NameNodeFile nnf, long txid, Canceler canceler)
```

```

> prog.beginPhase(Phase.SAVING_CHECKPOINT)
> if (storage.getNumStorageDirs(NameNodeDirType.IMAGE) == 0) { //没有 IMAGE 文件目录
>+ throw new IOException("No image directories available!");
> }
> if (canceler == null) canceler = new Canceler()
> SaveNamespaceContext ctx = new SaveNamespaceContext(source, txid, canceler)
    //创建用于 SaveNamespace 的 Context
>> this.sourceNamesystem = sourceNamesystem
>> this.txid = txid
>> this.canceller = canceller
> List<Thread> saveThreads = new ArrayList<Thread>() //创建一个用于 Thread 的 List
> for (Iterator<StorageDirectory>
    it = storage.dirIterator(NameNodeDirType.IMAGE); it.hasNext();) {
    //对于每个 IMAGE 文件目录(通常只有一个)
>+ StorageDirectory sd = it.next()
>+ FSImageSaver saver = new FSImageSaver(ctx, sd, nnf) //创建一个 FSImageSaver
    //参数 sd 各不相同
>+ Thread saveThread = new Thread(saver, saver.toString())
    //创建一个线程来执行这个 FSImageSaver
>+ saveThreads.add(saveThread) //将此线程放入 saveThreads 这个 List
>+ saveThread.start() //启动这个线程,执行 FSImageSaver.run()
> } //end for,每个类型为 Image 的目录都有一个线程照看
> waitForThreads(saveThreads) //等待这些线程全部完成
> saveThreads.clear()
> renameCheckpoint(txid, NameNodeFile.IMAGE_NEW, nnf, false)
    // 将 IMAGE_NEW 文件改名为 IMAGE 文件
>> for (StorageDirectory sd : storage.dirIterable(NameNodeDirType.IMAGE)) {
>>+ renameImageFileInDir(sd, fromNnf, toNnf, txid, renameMD5)
>>+> File fromFile = NNStorage.getStorageFile(sd, fromNnf, txid)
>>+> File toFile = NNStorage.getStorageFile(sd, toNnf, txid)
>>+> fromFile.renameTo(toFile)
>>+> if (renameMD5) MD5FileUtils.renameMD5File(fromFile, toFile)
>> }
> purgeOldStorage(nnf)
> ctx.markComplete()
> prog.endPhase(Phase.SAVING_CHECKPOINT)

```

对于每个存放映像文件的目录,包括 IMAGE 目录和既可用于 IMAGE 也可用于 EDITS 的共享目录,分别创建一个 FSImageSaver 线程,让此线程承担 saveFSImage 的工作:

```
class FSImageSaver implements Runnable {}
```

```

] SaveNamespaceContext context
] StorageDirectory sd //每个 FSImageSaver 线程的存储目录各不相同
] NameNodeFile nnf //文件类型,例如 IMAGE、EDITS、IMAGE_NEW
] run()
    > saveFSImage(context, sd, nnf) == FSImage.saveFSImage(context, sd, nnf)

```

这里 FSImageSaver 线程唯一的工作就是调用 FSImage.saveFSImage():

```
[FSImageSaver.run() > FSImage.saveFSImage()]
```

```

FSImage.saveFSImage(SaveNamespaceContext context, StorageDirectory sd,
                    NameNodeFile dstType)

> txid = context.getTxId() //见前面的 SaveNamespaceContext()
> File newFile = NNStorage.getStorageFile(sd, NameNodeFile.IMAGE_NEW, txid)
>> return new File(sd.getCurrentDir(), String.format("%s_%019d", type.getName(), imageTxId))
    //新文件的 type 为 NameNodeFile.IMAGE_NEW,
    //文件名为“fsimage.ckpt”,后随 19 位数字的 TxId
> File dstFile = NNStorage.getStorageFile(sd, dstType, txid)
    //目标文件的 type 为 NameNodeFile.IMAGE
    //文件名为“fsimage”,后随 19 位数字的 TxId
> FSImageFormatProtobuf.Saver saver = new FSImageFormatProtobuf.Saver(context)
> FSImageCompression compression = FSImageCompression.createCompression(conf)
> saver.save(newFile, compression) //保存在类型为 IMAGE_NEW 的 newFile 中
    == FSImageFormatProtobuf.Saver.save(newFile, compression)
>> FileOutputStream fout = new FileOutputStream(file)
>> fileChannel = fout.getChannel()
>> saveInternal(fout, compression, file.getAbsolutePath().toString())
>> fout.close()
> MD5FileUtils.saveMD5File(dstFile, saver.getSavedDigest())
> storage.setMostRecentCheckpointInfo(txid, Time.now())

```

保存文件系统映像意味着需要把内存中的许多对象写入磁盘,这里涉及对象的串行化。Hadoop 代码中提供了一个 FSImageFormatProtobuf 类,用来以 Protobuf 的格式存取文件系统映像。FSImageFormatProtobuf 中有个 Saver,还有个 Loader,这里用的是它的 Saver。有了这个,就无须关心这些对象的串行化,而可以把注意力集中在更为实质的那些操作上,那都在 saveInternal() 里面。

至此,我们还只看到要把文件系统映像保存到什么地方,但是却没有看到究竟保存些什么内容,这些内容是从哪里来的。我们来看 saveInternal() 的摘要:

```
[FSImageSaver.run() > saveFSImage() > FSImageFormatProtobuf.Saver.save() > saveInternal()]
```

```
saveInternal(FileOutputStream fout, FSImageCompression compression, String filePath)
```

```
> StartupProgress prog = NameNode.getStartupProgress()
```



```

> MessageDigest digester = MD5Hash.getDigester()    //用于 MDS
> underlyingOutputStream = new DigestOutputStream(
    new BufferedOutputStream(fout), digester)
> underlyingOutputStream.write(FSImageUtil.MAGIC_HEADER)
> fileChannel = fout.getChannel()
> FileSummary.Builder b = FileSummary.newBuilder()
    .setOndiskVersion(FSImageUtil.FILE_VERSION)
    .setLayoutVersion(NameNodeLayoutVersion.CURRENT_LAYOUT_VERSION)
> codec = compression.getImageCodec()    //也许需要压缩
> if (codec != null) {
>+ b.setCodec(codec.getClass().getCanonicalName())
>+ sectionOutputStream = codec.createOutputStream(underlyingOutputStream)
> } else {
>+ sectionOutputStream = underlyingOutputStream
> }
> saveNameSystemSection(b)    //保存该文件系统的种种参数
>> FSNamesystem fsn = context.getSourceNamesystem()
>> OutputStream out = sectionOutputStream
>> NameSystemSection.Builder b = NameSystemSection.newBuilder()
    .setGenstampV1(fsn.getGenerationStampV1())
    .setGenstampV1Limit(fsn.getGenerationStampV1Limit())
    .setGenstampV2(fsn.getGenerationStampV2())
    .setLastAllocatedBlockId(fsn.getLastAllocatedBlockId())
    .setTransactionId(context.getTxId())
>> b.setNamespaceId(fsn.unprotectedGetNamespaceInfo().getNamespaceID())
>> if (fsn.isRollingUpgrade()) {
>>+ b.setRollingUpgradeStartTime(fsn.getRollingUpgradeInfo().getStartTime())
>> }
>> NameSystemSection s = b.build()
>> s.writeDelimitedTo(out)
>> commitSection(summary, SectionName.NS_INFO)
    // Check for cancellation right after serializing the name system section.
    // Some unit tests, such as TestSaveNamespace# testCancelSaveNameSpace
    // depends on this behavior.
> context.checkCancelled()
-----
> Step step = new Step(StepType.INODES, filePath)
> prog.beginStep(Phase.SAVING_CHECKPOINT, step)
> saveInodes(b)    //保存索引节点
>> saver = new FSImageFormatPBINode.Saver(this, summary)

```

```

>> saver.serializeINodeSection(sectionOutputStream)
    == FSImageFormatPBINode.serializeINodeSection(sectionOutputStream)
                                                    //这是 INodeMap
>> saver.serializeINodeDirectorySection(sectionOutputStream) //这是 FSDirectory
>> saver.serializeFilesUCSection(sectionOutputStream) //这是 UnderConstruction 的文件节点
> saveSnapshots(b) //保存目录快照
>> snapshotSaver = new FSImageFormatPBSnapshot.Saver(
    this, summary, context, context.getSourceNamesystem())
>> snapshotSaver.serializeSnapshotSection(sectionOutputStream)
>> snapshotSaver.serializeSnapshotDiffSection(sectionOutputStream)
>> snapshotSaver.serializeINodeReferenceSection(sectionOutputStream)
> prog.endStep(Phase.SAVING_CHECKPOINT, step)
-----
> step = new Step(StepType.DELEGATION_TOKENS, filePath)
> prog.beginStep(Phase.SAVING_CHECKPOINT, step)
> saveSecretManagerSection(b) //属于文件访问安全机制
> prog.endStep(Phase.SAVING_CHECKPOINT, step)
-----
> step = new Step(StepType.CACHE_POOLS, filePath)
> prog.beginStep(Phase.SAVING_CHECKPOINT, step)
> saveCacheManagerSection(b)
> prog.endStep(Phase.SAVING_CHECKPOINT, step)
-----
> saveStringTableSection(b)
    // We use the underlyingOutputStream to write the header. Therefore flush
    // the buffered stream (which is potentially compressed) first.
> flushSectionOutputStream()
> FileSummary summary = b.build()
> saveFileSummary(underlyingOutputStream, summary)
> underlyingOutputStream.close()
> savedDigest = new MD5Hash(digester.digest())

```

由此可见, saveFSImage() 所保存的确实是关于一个 HDFS 的所有的元数据, 是整个 HDFS 的结构信息, 里面既包括相关的参数, 例如版本号, 也包括由 INode 构成的目录树和快照, 还包括与文件安全有关的内容。但是没有保存 EditLog, 那是需要另外保存的。

另外, 从代码中可见, 一个文件系统映像包含着好多内容。

首先是文件系统的种种参数, 尤其是版本号, 这些参数构成一个 NameSystemSection。

然后是一个 checkpoint。这个词带有“关卡”、“里程碑”、“基准点”的意思, 代表着此时此刻的文件系统映像, 将来如有必要还可以退回到这个映像。从代码摘要中可以看出, 文件系统的 checkpoint 是由两部分信息构成的: 一是与索引节点有关的信息, 这是通过 saveInodes() 保存的; 二是对于文件系统目录树上若干子树的“快照”, 即 Snapshot, 这是通过 saveSnapshots() 保存的。

还有就是有关访问权限和安全机制的信息、有关缓存管理的信息等。

由 `saveFSImage()` 保存下来的文件属于 `IMAGE_NEW` 文件。

这里对“快照”即 `Snapshot` 这个机制要做点说明。`Snapshot` 是对文件系统中个别子树即目录的“快照”，就是把一个目录在某个时间点上的状态复制记录下来。注意这跟 `checkpoint` 不同，与 `EditLog` 也没有什么关系（快照的创建操作本身倒是需要由 `EditLog` 加以记录），其作用相当于按目录进行文件备份。按常理，需要加以复制记录的不仅仅是目录中的那些 `Inode`，还应该包括所有的数据块，因为即使一个文件所含的数据块序列维持不变，这些数据块的内容也可能会有改变。但是，对于 HDFS 情况就不同了，在 HDFS 中，文件中的数据一经写入便不可修改，新的内容只能添加在文件的末尾，文件长度只能单调增长。在这样的条件下，为 HDFS 做 `Snapshot` 就简单了，只要把目录当时的所有 `Inode` 复制保存一个复份就可以了。要为一个目录保存快照时，首先要将这个目录设置成允许创建快照，即 `Snapshottable`，然后才可以创建快照。HDFS 会在这目录下创建一个名叫“.snapshot”的子目录，而创建的 `Snapshot` 则以“s0”、“s1”等为文件名存储在这个子目录下。所以，`snapshot` 中记载的是某个目录在既往某个时间点上的历史状况。

回到前面 `saveFSImageInAllDirs()` 的代码中，在所有这些线程都完成了 `saveFSImage()` 以后，有个 `renameCheckpoint()` 操作，把视作 `checkpoint` 的 `IMAGE_NEW` 文件改成 `IMAGE` 文件。如前所述，这两种文件的文件名编码是不一样的。

明白了文件系统映像的保存，即 `saveNamespace()` 和 `saveFSImage()`，读者自然不难理解，文件系统的 `format()` 只是在宿主文件系统中创建相关的目录。而文件系统映像的装载，即 `loadFromDisk()` 和 `loadFSImage()`，如果不考虑版本升级和故障恢复，那就只是 `saveNamespace()` 和 `saveFSImage()` 的逆向过程而已。

11.6 文件系统更改记录 `FSEditLog`

一个文件系统，只要不是严格的只读，在使用的过程中总是可能要发生变化的。比方说，在 HDFS 中，我们要在一个文件的末尾添加 20 个字节，于是我们就把这 20 个字节添在最后的那个块里，同时在 `NameNode` 上代表着这个文件的目录节点 (`Inode`) 中修改其文件长度，使它比原来增加 20 个字节，以反映实际情况，这就构成了对文件系统的一次更改。在正常的条件下，这个过程看似很严密。

可是，写在 `Inode` 中的信息是在 `NameNode` 的内存中，那是不可能每有更改都持久存储的，那样根本就不现实，因为 `Inode` 中的信息并非单独持久存储，要持久存储就是通过 `saveFSImage()` 进行整个文件系统映像的持久存储，那就是一个 `checkpoint`。现在，设想 `NameNode` 在下个 `checkpoint` 之前断电停机了，事实上这是随时都有可能发生的。下次开机的时候，`NameNode` 只能加载上一次 `checkpoint` 中的文件系统映像，于是那 20 个字节就丢失了。当然，那 20 个字节还在 `DataNode` 的磁盘上，但是 `NameNode` 告诉我们的文件长度却比实际长度短了 20 个字节。

怎么解决这个问题？现在广泛采用的就是把每次对文件系统的更改都用尽可能精炼的形式记录下来，并做持久存储，这样的记录就称为 `EditLog`，也称 `Journal`，相当于一个账本。这样，万一发生故障，`NameNode` 就可以在前一个 `checkpoint` 的基础上根据 `EditLog` 中发生于前

一个 checkpoint 之后的那些记载重演一遍,使文件系统中的元数据与实际情况相符。当然,如果 NameNode 在关机前夕形成了一个 checkpoint,下次开机时所用的是最新的文件系统映像,并且 EditLog 中没有产生于那次 checkpoint 之后的记录,这时候文件系统中的元数据与实际情况相符,那就无须重演了。

这样的问题和解决方案,其实在单机中也存在。在 Linux 上,我们在关机之前要打一条 sync 命令,就是为了把内存中的文件系统映像写回磁盘,那就相当于一次 saveFSImage()。平时 Linux 也会定时自动执行 sync,但是如果意外重启,文件系统就可能陷入不一致的状态,这时候就得试图通过 fsck 进行修复,但是因为没有这样的账本,修复就得不到保证。所以,对于一些所谓“Mission Critical”的应用,例如银行等,也要采用 Journal 的方法。后来 Linux 内核中还实现了 Journal 文件系统,即采用 Journal 技术的文件系统。所以 Hadoop 的 EditLog 与单机中 Journal 文件系统的思路是一致的。

原理似乎并不复杂,问题是在 Hadoop 中具体怎么实现。
先简单说一下日志信息的存储。Hadoop 的日志信息可以作为文件存储在宿主机的文件系统中,也可以在集群中专门安排一个节点作为日志服务器,提供存储日志信息的服务。

前面说过,日志记录的信息要尽量精简,具体要记录些什么得要看所记录的操作对于文件系统改变了什么。Hadoop 源码中定义了一个关于文件操作的枚举类型作为操作代码,这就好像计算机指令中的操作码:

```
enum FSEditLogOpCodes {
    OP_ADD                                ((byte) 0),
    OP_RENAME_OLD                         ((byte) 1), // deprecated operation
    OP_DELETE                             ((byte) 2),
    OP_MKDIR                              ((byte) 3),
    OP_SET_REPLICATION                    ((byte) 4),

    OP_SET_PERMISSIONS                    ((byte) 7),
    OP_SET_OWNER                           ((byte) 8),
    OP_CLOSE                              ((byte) 9),
    ...

    OP_END_LOG_SEGMENT                    ((byte) 23),
    OP_START_LOG_SEGMENT                   ((byte) 24),
    OP_UPDATE_BLOCKS                       ((byte) 25),
    OP_CREATE_SNAPSHOT                     ((byte) 26),
    OP_DELETE_SNAPSHOT                     ((byte) 27),
    ...

    OP_ADD_BLOCK                           ((byte) 33),
    ...

    OP_SET_ACL                             ((byte) 40),
    OP_ROLLING_UPGRADE_START               ((byte) 41),
    OP_ROLLING_UPGRADE_FINALIZE            ((byte) 42),
```

```

...
OP_SET_STORAGE_POLICY          ((byte) 45),
// Note that the current range of the valid OP code is 0~127
OP_INVALID                      ((byte) - 1)
}

```

凡是对 HDFS 文件系统的结构、内容、状态造成某种改变的操作,全都在这里了。注意,这里定义了操作码的数据类型是 byte,只占一个字节。

但是光有一个操作码显然不够,还得要有对于具体内容的说明,就像计算机指令一样,有了操作码还要有操作数。对于文件操作的记录当然有共性,从而可以有公共的格式,所以 Hadoop 定义了一个抽象类 FSEditLogOp,作为所有操作记录的基础,然后针对不同的操作可以扩充这个抽象类,加上不同的信息:

```

abstract class FSEditLogOp {}
] FSEditLogOpCodes opCode
] long txid
] byte[] rpcClientId
] int rpcCallId
- - - - - 以上为公共的数据结构部分 - - - - -
] abstract class AddCloseOp extends FSEditLogOp implements BlockListUpdatingOp {}
] static class AddBlockOp extends FSEditLogOp {} //这是对 OP_ADD_BLOCK 的操作记录
]] String path                                //增加了 path 等三项信息
]] Block penultimateBlock
]] Block lastBlock
] static class CloseOp extends AddCloseOp {}
] static class DeleteOp extends FSEditLogOp {}
] static class SetOwnerOp extends FSEditLogOp {} //这是对 OP_SET_OWNER 的操作记录
]] String src                                //增加了 src、username 等三项信息
]] String username
]] String groupname
] static class LogSegmentOp extends FSEditLogOp {}
] static class CreateSnapshotOp extends FSEditLogOp {}
] ...
] static class OpInstanceCache {}
]] EnumMap<FSEditLogOpCodes, FSEditLogOp> inst =
    new EnumMap<FSEditLogOpCodes, FSEditLogOp>(FSEditLogOpCodes.class)
]] OpInstanceCache()
> inst.put(OP_ADD, new AddOp()) //对每一种具体的 FSEditLogOp 都是这样,下同
> ... //形成一个 EnumMap,凭操作码就可从中找到相应的操作记录对象,以供使用
] static class AclEditLogUtil {}

```

这个抽象类的数据结构部分包括 opCode、txid 等 4 项数据,其余是对操作和内嵌类的定

义。这个抽象类有点特殊：对各种具体操作记录的定义，即对于抽象类 `FSEditLogOp` 的扩充全在这抽象类内部，这些成分都定义成了静态(static)成分。

这里还有一个静态成分 `OpInstanceCache`，则用来形成一个 `EnumMap`，相当于一个二维表，以后凭 `OP_ADD` 之类的操作码就可从中找到预先创建好了的相应操作记录对象，例如 `AddOp` 对象。这样，每当需要记录某种操作而要用到某类对象的时候，就不必每次都临时创建该类操作记录对象，只要从这个表中拿过来填写上几个具体数据就行。这是一种编程和优化的技巧。

如果手中有了一条操作记录需要进入日志，程序中该怎么处理呢？Hadoop 在 `FSImage` 内部提供了一个 `FSEditLog` 类对象来做这个事，下面是这个类的摘要：

```
class FSEditLog implements LogsPurgeable {
    ] State state //日志的状态,见下面的枚举类型定义
    ] JournalSet journalSet //一组 JournalManager,管理着各自的日志存储介质
    ] EditLogOutputStream editLogStream //EditLog 的输出通道,通向具体的 Journal
    ] long txid = 0 //a monotonically increasing counter that represents transactionIds.
    ] long synctxid = 0 //stores the last synced transactionId.
    ] long curSegmentTxId = HdfsConstants.INVALID_TXID
    ] long numTransactions; // number of transactions
    ] NNStorage storage //相关的 NNStorage,见前述
    ] Configuration conf
    ] List<URI> editsDirs //用于日志的目录路径
    ] List<URI> sharedEditsDirs
    ] ThreadLocal<OpInstanceCache> cache = new ThreadLocal<OpInstanceCache>()
        ] OpInstanceCache initialValue()
        > return new OpInstanceCache()
    ] static ThreadLocal<TransactionId> myTransactionId = new ThreadLocal<TransactionId>()
    - - - - - 以上为数据结构部分,以下提供种种日志记录的操作方法 - - - - -
    ] logOpenFile(String path, INodeFile newNode, boolean overwrite, boolean toLogRpcIds)
    ] logCloseFile(String path, INodeFile newNode)
    ] logAddBlock(String path, INodeFile file)
    ] ...
    ] logMkdir(String path, INode newNode)
    ] ...
    - - - - - 以下是其他的类型或方法定义 - - - - -
    ] enum State {UNINITIALIZED, BETWEEN_LOG_SEGMENTS,
        IN_SEGMENT, OPEN_FOR_READING, CLOSED;}
    ] class TransactionId {}
    ] openForWrite()
        > Preconditions.checkState(state == State.BETWEEN_LOG_SEGMENTS,
            "Bad state: %s", state)
        > segmentTxId = getLastWrittenTxId() + 1
```



```

> List<EditLogInputStream> streams = new ArrayList<EditLogInputStream>()
> journalSet.selectInputStreams(streams, segmentTxId, true)
> startLogSegment(segmentTxId, true)
> assert state == State.IN_SEGMENT : "Bad state: " + state
] formatNonFileJournals(NamespaceInfo nsInfo)
    //Format all configured journals which are not file-based.
    //File-based journals are skipped, since they are formatted by the Storage format code.
] logEdit(final FSEditLogOp op) //Write an operation to the edit log.
    //Do not sync to persistent store yet.
] beginTransaction() //开始一次日志写入
] endTransaction(long start) //结束一次日志写入
] logSync() //对缓冲着的日志进行同步
] startLogSegment(final long segmentTxId, boolean writeHeaderTxn) //打开一个日志段
] endCurrentLogSegment(boolean writeEndTxn) //关闭当前日志段
] purgeLogsOlderThan(final long minTxIdToKeep) //冲洗掉日志中过老的记录
] getJournalClass(Configuration conf, String uriScheme) //获取用作 Journal 的类
] createJournal(URI uri) //创建 Journal 对象

```

这里有个特殊的成分 `cache`, 这是个 `ThreadLocal<OpInstanceCache>`, 就是属于线程本地的 `OpInstanceCache`。属于同一进程的线程共享同一个存储空间, 能为其中一个线程访问使用的对象, 别的线程也就都能访问使用, 这样就可能互相干扰。所以有时候我们需要有让一个线程单独使用的存储空间, 称为 TLS, 即“Thread Local Storage”, 在 Java 虚拟机上就是 `ThreadLocal`。这里的 `cache` 就是一个 `ThreadLocal` 的 `OpInstanceCache` 对象。前面讲到, 为了避免每次要写日志时都临时创建特定的 `FSEditLogOp` 对象, 作为一种优化, 代码中预先创建了全套的 `FSEditLogOp` 对象, 放在一个 `EnumMap` 中, 需要用时凭操作码找到相应的对象, 填上当时实际的参数就行。但是, 这种预先创建的 `FSEditLogOp` 对象必须让每个有日志要求的线程都有自己的一份才行, 否则就会互相“打架”, 所以必须用 `ThreadLocal`, 让这个线程看到的 `cache` 与那个线程看到的 `cache` 各不一样, 做到“桥归桥, 路归路”。

这里还有个 `myTransactionId` 也是 `ThreadLocal` 的变量, 这是当一个线程写入日志时用来记录当时的 Transaction ID 的。这就好像你投寄一份快件, 人家就给你一个号码, 以便查询。但是, 如果所有的线程都共用同一个变量, 那就又互相“打架”了。所以, 像这样的变量, 应该是形式上、概念上只有一个, 以便代码的编写。而实际上每个线程都有自己的一份, 以免干扰。这就是 `ThreadLocal` 变量的作用。

`FSEditLog` 类对象属于 `FSImage`, 是 `FSImage` 内部的一个成分。这很好理解, 因为这个日志所记录的就是对于具体文件系统映像的改变。`FSEditLog` 对于前面定义的每一种操作都提供一个方法函数, 例如对于创建目录就有 `logMkDir()`, 对于打开文件就有 `logOpenFile()`, 余可类推。这样, 例如对于打开文件就只要调用 `FSEditLog.logOpenFile()`, 就可以把一次具体的打开文件操作记入日志。

除这些以外还定义了好多供内部使用的方法。

日志中的记录是分段 (Segment) 的, 任何操作记录都必须在某一个段中, 所以一定要在

startLogSegment()以后才可以往里面写,一旦当前段关闭,就必须在下次打开以后才能再写。所以,FSEditLog 的状态就是例如 BETWEEN_LOG_SEGMENTS、IN_SEGMENT 等。还有,对日志的写入理应是不可分割的“原子操作”,但是实际上当然办不到,所以要把对日志的写入做成一次 Transaction,每次写入的前后要分别调用 beginTransaction()和 endTransaction()。

介绍完有关日志的这些要素,我们得用一种情景把它们串起来,在 Hadoop 源代码中找一个尽量简单却又反映全貌的实例考察其具体实现。这里用的实例是 setOwner(),就是设置一个文件的文件主和用户组,这是现代文件系统的一项必备功能,但是很简单。我们跳过命令行和 RPC,直接从对于文件系统的调用开始:

```
FSNamesystem.setOwner(String src, String username, String group)
    //src 是文件路径,username 是新的文件主,group 是用户组
> setOwnerInt(src, username, group) //Int 是 Internal 的意思
>> String src = srcArg
>> FSPermissionChecker pc = getPermissionChecker()
>> ... //访问权限控制,此处从略
>> dir.setOwner(src, username, group)
    == FSDirectory.setOwner(String src, String username, String groupname)
>>> unprotectedSetOwner(src, username, groupname) //实际的操作由这个函数完成
>> getEditLog().logSetOwner(src, username, group) //记录对于文件系统的此项更改
    == FSEditLog.logSetOwner(src, username, group) //这要通过 FSEditLog 完成
>> getEditLog().logSync() //同步日志记录,要求将日志写入持久存储介质,如磁盘
>> logAuditEvent(true, "setOwner", srcArg, null, resultingStat)
    == FSNamesystem.logAuditEvent(...)
```

真正的设置文件主操作是由 FSDirectory.setOwner()实现的,其内部起实质作用的是 unprotectedSetOwner()。本来,完成了对 FSDirectory.setOwner()的调用,目的就已达到,但是现在加上了对 FSEditLog.logSetOwner()的调用,要把本次对于文件系统的操作记入日志。

```
[FSNamesystem.setOwner() > FSEditLog.logSetOwner()]
```

```
void logSetOwner(String src, String username, String groupname) {
    SetOwnerOp op = SetOwnerOp.getInstance(cache.get())
        .setSource(src).setUser(username).setGroup(groupname);
    logEdit(op);
}
```

把这一段源代码改写成我们的摘要,就是这样:

```
logSetOwner(String src, String username, String groupname)
> op = SetOwnerOp.getInstance(cache.get())
    //注意 cache 是 ThreadLocal,每个线程都有自己的 cache
>> return (SetOwnerOp)cache.get(OP_SET_OWNER) //以操作码 OP_SET_OWNER 为依据
    //从前述的 EnumMap 中抓到预先创建备用的 SetOwnerOp 对象
```

```
> op.setSource(src) //对此 SetOwnerOp 对象设置参数
> op.setUser()
> op.setGroup()
> logEdit(op)
```

如前所述,为提高效率,系统预先创建了整套的 FSEditLogOp 对象存放在一个 EnumMap 中,需要的时候按操作码,在这里是 OP_SET_OWNER,就可找到相应的 FSEditLogOp 对象,在这里就是 SetOwnerOp 对象。为避免线程间互相干扰,这个 EnumMap 是 ThreadLocal,每个线程都有自己的一套。这样,拿到属于自己的 SetOwnerOp 对象以后,填写好了参数就可以调用 logEdit()写入日志:

```
[FSNamesystem.setOwner() > FSEditLog.logSetOwner()>logEdit()]
```

```
FSEditLog.logEdit(final FSEditLogOp op)
> synchronized (this) { //进入临界区,加锁
>> waitIfAutoSyncScheduled() //wait if an automatic sync is scheduled
>>> while (isAutoSyncScheduled) this.wait(1000)
>> long start = beginTransaction()
>> op.setTransactionId(txid)
>>> txid++
>>> TransactionId id = myTransactionId.get()
>>> id.txid = txid
>>> return now()
>> editLogStream.write(op) == EditLogOutputStream.write(op) //经输出通道写入 Journal
>> endTransaction(start)
>>> long end = now()
>>> numTransactions++
>> if (!shouldForceSync()) return
>> isAutoSyncScheduled = true
> } //离开临界区,解锁
> logSync() //sync buffered edit log entries to persistent store
```

注意,这里的操作是在一个 synchronized 临界区中进行的,操作前加了锁,操作完成后才解锁。也就是说,如果有一个线程的程序进入了这个区间,别的线程就进不来。哪怕这个线程本身也因为对 editLogStream 的写操作被阻塞(例如写磁盘,或网络发送尚未得到回应)而睡眠等待,别的线程也只好在这个临界区外等待。这样,就保证了线程间的互斥,即在任何时候,在 NameNode 上至多只有一个线程在执行 logEdit()中除最后一步 logSync()之外的那些操作。Hadoop 的代码中类似于这样的临界区不在少数,但是一般我们加以省略,以免冲淡对于“主旋律”的注意,但是在这个地方就不能省略了。

即便进入了这个临界区,这里也还有另一种互斥,就是如果 isAutoSyncScheduled 为 true,就必须等待其变成 false 才能往下走,而在离开这个临界区前夕则又将此变量设置成 true。

那么是谁把这个变量 isAutoSyncScheduled 设置成 false 呢? 我们继续往下看:

```
[FSNamesystem.setOwner() > FSEditLog.logSetOwner()>logEdit()>logSync()]
```

FSEditLog.logSync()

```
> long mytxid = myTransactionId.get().txid
> boolean sync = false
> synchronized (this) { //进入临界区,加锁
>+ while (mytxid > synctxid && isSyncRunning) wait(1000)
//if somebody is already syncing, then wait
//这里,mytxid是我要冲刷的那份记录的txid,synctxid是已经冲刷的记录的txid
>+ if (mytxid <= synctxid) { //If this transaction was already flushed, then nothing to do
//我的号已经过了,不用再要求冲刷(flush)了
>++ numTransactionsBatchedInSync++
>++ return
>+ }
>+ syncStart = txid //now, this thread will do the sync
>+ isSyncRunning = true
>+ sync = true
>+ if (journalSet.isEmpty()) throw new IOException("No journals available to flush")
>+ editLogStream.setReadyToFlush() == EditLogOutputStream.setReadyToFlush()
//swap buffers,双缓冲互换,为flush做好准备
>+ doneWithAutoSyncScheduling()
>+> if (isAutoSyncScheduled) {
>+>+ isAutoSyncScheduled = false //允许进入logEdit()临界区的线程前行
>+>+ notifyAll() //唤醒正在等待的线程
>+> }
>+ logStream = editLogStream
> } //离开临界区,解锁
> long start = now()
> if (logStream != null) logStream.flush()
>> numSync++
>> long start = now()
>> flushAndSync(durable) //调用具体输出通道的flushAndSync()
>> long end = now()
>> totalTimeSync += (end - start)
> long elapsed = now() - start
> synchronized (this) { //进入临界区,加锁
>+ if (sync) {
>++ synctxid = syncStart
>++ isSyncRunning = false
>+ }
```

```
>+ this.notifyAll()      //唤醒正在等待的线程
> } //离开临界区,解锁
```

原来,把 `isAutoSyncScheduled` 设置成 `false` 的仍是同一个线程。这是在干什么呢? 原来,这里的前后三个临界区,是各有深意的。当前线程把一条操作记录写入日志并非一步就能完成,第一步是通过 `EditLogOutputStream.write()` 写入输出通道的缓冲区,写的时候当然不能让别的线程打扰(如果它也要写),这是第一个临界区的作用,在前面的 `logEdit()` 中。然后,还需要把缓冲区的内容冲刷(`flush`)到 `Journal`,即持久存储的介质上,这一步理应也在临界区中进行,但是这一步所需的时间可能较长,如果都放在临界区中,加锁的时间就可能太长了。所以这里采用了一种“双缓冲”的办法,就是搞两个缓冲区交替使用,当一个缓冲区的内容正在冲刷出去的时候,另一个缓冲区可以让别的线程写入;然后交换一下,冲刷已经写入的那个缓冲区,而已经冲刷的那个缓冲区则又可写入了,就像“跷跷板”一样。这样,只要把交换缓冲区的操作,即这里的 `setReadyToFlush()` 放在临界区里就可以了。所以,一旦通过 `setReadyToFlush()` 交换了缓冲区,就可以把 `isAutoSyncScheduled` 设置成 `false`,让别的线程进来写缓冲区了,这就是第二个缓冲区的作用。至于第三个缓冲区,则是对改变两个共享变量的互斥保护。

读者也许觉得这过程搞得太复杂了,一下子难以理解。但是多读这样的代码可以锻炼我们的思维,一方面使之更加缜密,另一方面也更懂得怎样精益求精。事实上,只用一个临界区把所有这些操作全都包起来也并非不可以,但那样对于性能会有负面的影响。

最后还有个问题,就是这个日志的持久存储,即 `Journal`,还有这个日志输出通道 `EditLogOutputStream` 是怎么来的。

我们在前面 `FSEditLog` 类的摘要中看到,里面有个 `JournalSet` 类的对象 `journalSet`。显然这是一个 `Journal` 对象的集合。这个集合是在什么时候创建的呢? 首先,在启动一个全新的 `NameNode` 而进行 HDFS 的格式化的时候,要创建一个 `FsImage` 对象,而 `journalSet` 是里面的一个成分、一个部件,所以此时会调用 `fsImage.getEditLog().initJournalsForWrite()`,那就是 `FSEditLog.initJournalsForWrite()`,那里面会创建 `JournalSet`。这是其原始的来源。其次,在此后每次启动 `NameNode`,在其初始化阶段会通过 `loadFSImage()` 加载保存在磁盘上的文件系统映像,在此过程中会调用一个函数 `initEditLog()`,这里面也会辗转创建 `JournalSet`。其实还有别的可能,但这二者是主要的。我们就从 `initEditLog()` 看起。

```
[NameNode.initialize() > loadNamesystem() > FSNamesystem.loadFromDisk() >
FSNamesystem.loadFSImage() > FSImage.recoverTransitionRead() > FSImage.loadFSImage()
> FSImage.initEditLog()]
```

```
FSImage.initEditLog(StartupOption startOpt)
> String nameserviceId = DFSUtil.getNamenodeNameServiceId(conf)
> if (!HAUtil.isHAEnabled(conf, nameserviceId)) {
>+ editLog.initJournalsForWrite() == FSEditLog.initJournalsForWrite()
>+> initJournals(this.editsDirs)
>+> state = State.BETWEEN_LOG_SEGMENTS
>+ editLog.recoverUnclosedStreams()
> } else ...
```

我们现在不关心 HA 模式,即“高可用(High Availability)”容错模式,这里仍是调用 `initJournalsForWrite()`,这跟 `format()`时一样。而 `initJournalsForWrite()`则调用 `initJournals()`:

```

FSEditLog.initJournals(List<URI> dirs) {
    > minimumRedundantJournals = conf.getInt(
        DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_MINIMUM_KEY,
        DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_MINIMUM_DEFAULT)
        //至少 1 个 JournalManager
    > journalSet = new JournalSet(minimumRedundantJournals)
    > for (URI u : dirs) {
    >+ boolean required = FSNamesystem.getRequiredNamespaceEditsDirs(conf).contains(u)
    >+> Set<URI> ret = new HashSet<URI>()
    >+> ret.addAll(getStorageDirs(conf, DFS_NAMENODE_EDITS_DIR_REQUIRED_KEY))
        // “file:///tmp/hadoop/dfs/name”
    >+> ret.addAll(getSharedEditsDirs(conf))
    >+> return ret
    >+ if (u.getScheme().equals(NNStorage.LOCAL_URI_SCHEME)) {
    >++ StorageDirectory sd = storage.getStorageDirectory(u)
    >++ if (sd != null) {
    >+++ journalManager = new FileJournalManager(conf, sd, storage)
    >+++ journalSet.add(journalManager, required, sharedEditsDirs.contains(u))
        //把 FileJournalManager 对象放入集合 journalSet
    >++ }
    >+ } else {
    >++ journalManager = createJournal(u)
    >++> clazz = getJournalClass(conf, uri.getScheme())
    >++>> String key = DFSConfigKeys.DFS_NAMENODE_EDITS_PLUGIN_PREFIX + "." + uriScheme
        // name == "dfs.namenode.edits.journal-plugin.qjournal"
    >++>> clazz = conf.getClass(key, null, JournalManager.class)
        //value == "org.apache.hadoop.hdfs.qjournal.client.QuorumJournalManager"
    >++>> return clazz
    >++> cons = clazz.getConstructor(Configuration.class, URI.class, NamespaceInfo.class)
    >++> return cons.newInstance(conf, uri, storage.getNamespaceInfo())
        //创建 QuorumJournalManager 对象
    >++ journalSet.add(journalManager, required, sharedEditsDirs.contains(u))
        //把 QuorumJournalManager 对象放入集合 journalSet
    >+ }
    > } //end for

```

按配置文件中的设置,集合 `journalSet` 的大小一般是 1 或 2。

Hadoop 的代码中定义了一个界面 `JournalManager`,实现这个界面的类则有三种,就是

QuorumJournalManager、FileJournalManager 和 BackupJournalManager。这里,Journal 就是账本、日志的意思。

我们大致看一下这三个类的摘要,首先是 QuorumJournalManager,所谓 QuorumJournal 的“账本”在远程的一个或一组 JournalNodes 节点上,就好像一组服务器一样,QuorumJournalManager 就是对远程账本的管理者:

```
class QuorumJournalManager implements JournalManager {}
] URI uri
] AsyncLoggerSet loggers
] URLConnectionFactory connectionFactory
] outputBufferCapacity = 512 * 1024
] QuorumJournalManager(Configuration conf, URI uri,
                        NamespaceInfo nsInfo, AsyncLogger.Factory loggerFactory)
] startLogSegment(long txId, int layoutVersion)
  > QuorumCall<AsyncLogger, Void> q = loggers.startLogSegment(txId, layoutVersion)
  > loggers.waitForWriteQuorum(q, startSegmentTimeoutMs, "startLogSegment(" + txId + ")")
  > return new QuorumOutputStream(loggers, txId,
                                outputBufferCapacity, writeTxnsTimeoutMs)
```

再看 BackupJournalManager 类的摘要,BackupJournal 的“账本”也在另一个节点 BackupNode 上,NameNode 通过 RPC 将日志信息记入 BackupNode 上的账本:

```
class BackupJournalManager implements JournalManager {}
] NamenodeRegistration bnReg
] JournalInfo journalInfo
] BackupJournalManager(NamenodeRegistration bnReg, NamenodeRegistration nnReg)
] startLogSegment(long txId, int layoutVersion)
  > EditLogBackupOutputStream stm = new EditLogBackupOutputStream(bnReg, journalInfo)
  > stm.startLogSegment(txId)
  > return stm
```

还有 FileJournalManager 类的摘要,这是以宿主系统中的文件充当账本:

```
class FileJournalManager implements JournalManager {}
] StorageDirectory sd
] outputBufferCapacity = 512 * 1024
] FileJournalManager(Configuration conf, StorageDirectory sd,
                    StorageErrorReporter errorReporter)
] startLogSegment(long txid, int layoutVersion)
  > EditLogOutputStream stm =
    new EditLogFileOutputStream(conf, currentInProgress, outputBufferCapacity)
  > stm.create(layoutVersion)
  > return stm
```

```

] class EditLogFile {}
]] File file
]] long firstTxId, lastTxId

```

对于记录 EditLog 的三种不同手段,写 EditLog 的输出流也有所不同,所以 Hadoop 的代码中定义了几种不同的输出流:

```

abstract class EditLogOutputStream implements Closeable {}

class QuorumOutputStream extends EditLogOutputStream {}
class EditLogBackupOutputStream extends EditLogOutputStream {}
class EditLogFileOutputStream extends EditLogOutputStream {}
class JournalSetOutputStream extends EditLogOutputStream {}

```

这些输出流的目的,无非就是为了 EditLog 的持久存储,以防因为机器损坏或断电而引起的 EditLog 丢失。

11.7 FSEditLog 与 Journal

要是 FSEditLog 只存在于内存中而不加以持久存储,那显然是不安全的,一旦 EditLog 丢失,文件系统就可能陷入不一致的状态。

把 FSEditLog 持久存储下来,就称为 Journal,在内存中则以 Journal 对象为代表。一个 Journal 对象代表着 FSEditLog 的一份持久存储。

```

class Journal implements Closeable {}
] EditLogOutputStream curSegment
] String journalId
] JNStorage storage
] PersistentLongFile lastPromisedEpoch
] PersistentLongFile lastWriterEpoch
] BestEffortLongFile committedTxnId
] FileJournalManager fjm
] Journal(Configuration conf, File logDir, String journalId, StartupOption startOpt,
StorageErrorReporter errorReporter)
] journal(RequestInfo reqInfo, long segmentTxId, long firstTxnId, int numTxns, byte[] records)
] startLogSegment(RequestInfo reqInfo, long txid, int layoutVersion)
] finalizeLogSegment(RequestInfo reqInfo, long startTxId, long endTxId)
] syncLog(RequestInfo reqInfo, final SegmentStateProto segment, final URL url)

```

也可以在集群中安排一个节点专门做 FSEditLog 持久存储,这样的节点就是 JournalNode:

```

class JournalNode implements Tool, Configurable, JournalNodeMBean {}
] JournalNodeRpcServer rpcServer
] JournalNodeHttpServer httpServer

```

```

] Map<String, Journal> journalsById
] File localDir
] getOrCreateJournal(String jid, StartupOption startOpt)
    > QuorumJournalManager.checkJournalId(jid)
    > Journal journal = journalsById.get(jid)
    > if (journal == null) {
    >+ File logDir = getLogDir(jid)
    >+ journal = new Journal(conf, logDir, jid, startOpt, new ErrorReporter())
    >+ journalsById.put(jid, journal)
    > }
] start()    //Start listening for edits via RPC.
] doUpgrade(String journalId, StorageInfo sInfo)
    > getOrCreateJournal(journalId).doUpgrade(sInfo)
] doFinalize(String journalId)
    > getOrCreateJournal(journalId).doFinalize()
] doRollback(String journalId)
    > getOrCreateJournal(journalId, StartupOption.ROLLBACK).doRollback()
] main(String[] args)
    > jn = new JournalNode()
    > System.exit(ToolRunner.run(jn, args))

```

但是当然也可以“小打小闹”，就把 FSEditLog 存储在 NameNode 本地的磁盘上。

不管是存放在本地磁盘上还是专门的 JournalNode 上，或者别的什么地方，EditLog 都应被视作 FSImage 的一部分，缺了自从上一个 checkpoint，即 FSImage 硬拷贝之后的 EditLog，持久存储着的 FSImage 就不完整，就只是已经过时的 FSImage，所以在装载 FSImage 的时候一定要把 FSEditLog 一起装载进来。

```
[FSImage.loadFSImage() > loadEdits()]
```

```

FSImage.loadEdits(Iterable<EditLogInputStream> editStreams, FSNamesystem target,
                  StartupOption startOpt, MetaRecoveryContext recovery)
> StartupProgress prog = NameNode.getStartupProgress()
> prog.beginPhase(Phase.LOADING_EDITS)
> prevLastAppliedTxId = lastAppliedTxId
> loader = new FSEditLogLoader(target, lastAppliedTxId)
> // Load latest edits
> for (EditLogInputStream editIn : editStreams) { //Journal 可能分成好几个文件(好几个流)
>+ LOG.info("Reading " + editIn + " expecting start txid #" + (lastAppliedTxId + 1))
>+ loader.loadFSEdits(editIn, lastAppliedTxId + 1, startOpt, recovery)
//装载一个 Journal,把 EditLog 记录一条条装载进来
>+> FSImage.LOG.info("Start loading edits file " + edits.getName())

```

```

>+> numEdits = loadEditRecords(edits, false, expectedStartingTxId, startOpt, recovery)
>+> edits.close()
>+ lastAppliedTxId = loader.getLastAppliedTxId()
> }
> FSEditLog.closeAllStreams(editStreams) //装载完了,关闭输入流
> updateCountForQuota(target.dir.rootDir)
> prog.endPhase(Phase.LOADING_EDITS)
> return lastAppliedTxId - prevLastAppliedTxId

```

Hadoop 源码中的函数 loadEditRecords(), 可远远不只是“装载”一条条日志记录将其读进内存这么简单, 实际上还包括把 EditLog 的记录逐条在 FSImage 上重演:

```
[FSImage.loadFSImage() > loadEdits() > FSEditLogLoader.loadFSEdits() > loadEditRecords()]
```

```

FSEditLogLoader.loadEditRecords(EditLogInputStream in, boolean closeOnExit,
                                long expectedStartingTxId, StartupOption startOpt,
                                MetaRecoveryContext recovery)

> FSDirectory fsDir = fsNamesys.dir
> opCounts =
    new EnumMap<FSEditLogOpCodes, Holder<Integer>>> (FSEditLogOpCodes.class)
> recentOpcodeOffsets[] = new long[4]
> expectedTxId = expectedStartingTxId
> lastTxId = in.getLastTxId()
> numTxns = (lastTxId - expectedStartingTxId) + 1
> lastInodeId = fsNamesys.getLastInodeId()
> while (true) {
>+ FSEditLogOp op = in.readOp() //读入一个 FSEditLogOp, 即一条日志, 记录
>+ if (op == null) break //没有了, 就跳出循环
>+ recentOpcodeOffsets[(int)(numEdits % recentOpcodeOffsets.length)] = in.getPosition()
>+ if (op.hasTransactionId()) {
>++ if (op.getTransactionId() > expectedTxId) {
>+++ MetaRecoveryContext.editLogLoaderPrompt(
    "There appears to be a gap in the edit log." + ...)
>++ } else {
>+++ MetaRecoveryContext.editLogLoaderPrompt(
    "There appears to be an out-of-order edit in the edit log." + ...)
>+++ continue
>++ }
>+ } //end if (op.hasTransactionId())
>+ inodeId = applyEditLogOp(op, fsDir, startOpt, in.getVersion(true), lastInodeId)
    //将一条 EditLog 记录重演在 FSImage 上

```

```

>+ if (lastInodeId < inodeId) lastInodeId = inodeId
>+ incrOpCount(op.opCode, opCounts, step, counter)
>+ if (op.hasTransactionId()) {
>+ lastAppliedTxId = op.getTransactionId()
>+ expectedTxId = lastAppliedTxId + 1
>+ } else {
>+ expectedTxId = lastAppliedTxId = expectedStartingTxId
>+ }
>+ numEdits ++
>+ totalEdits ++
> } //end while
> fsNamesys.resetLastInodeId(lastInodeId)

```

所以,装载 EditLog 的关键是 applyEditLogOp(),是让 EditLog 记录在文件系统上重演。

11.8 EditLog 记录的重演

之所以要有 EditLog,就是为了可以在 FSImage 上重演。这样,对于 FSImage 就可以过一段(不短的)时间才做一个硬拷贝,因为那个开销太大;平时对 FSImage 的操作则随时记录在 FSEditLog 的硬拷贝中;万一发生事故,两个硬拷贝都还在,把记录在 EditLog 中的操作在(已经过时的)文件系统映像上重演一遍,就恢复过来了。所以操作的重演是关键:

```

[FSImage.loadFSImage() > loadEdits() > FSEditLogLoader.loadFSEdits() >
loadEditRecords() > applyEditLogOp()]

```

```

FSEditLogLoader.applyEditLogOp(FSEditLogOp op, FSDirectory fsDir,
                                StartupOption startOpt, int logVersion, long lastInodeId)
> long inodeId = INodeId.GRANDFATHER_INODE_ID
> boolean toAddRetryCache = fsNamesys.hasRetryCache() && op.hasRpcIds()
> switch (op.opCode) {
> case OP_ADD:
>+ AddCloseOp addCloseOp = (AddCloseOp)op
>+ String path = renameReservedPathsOnUpgrade(addCloseOp.path, logVersion)
>+ // There three cases here:
>+ // 1. OP_ADD to create a new file
>+ // 2. OP_ADD to update file blocks
>+ // 3. OP_ADD to open file for append
>+ // See if the file already exists (persistBlocks call)
>+ INodesInPath iip = fsDir.getINodesInPath(path, true)
>+ INode[] inodes = iip.getINodes()
>+ INodeFile oldFile = INodeFile.valueOf(inodes[inodes.length - 1], path, true)

```

```

>+ if (oldFile!= null && addCloseOp.overwrite) { // This is OP_ADD with overwrite
>++ fsDir.unprotectedDelete(path, addCloseOp.mtime)
>++ oldFile = null
>+ }
>+ INodeFile newFile = oldFile
>+ if (oldFile== null) { // this is OP_ADD on a new file (case 1),文件原先不存在
>++ replication = fsNamesys.getBlockManager().adjustReplication(addCloseOp.replication)
>++ assert addCloseOp.blocks.length == 0
>++ // add to the file tree
>++ inodeId = getAndUpdateLastInodeId(addCloseOp.inodeId, logVersion, lastInodeId)
>++ newFile = fsDir.unprotectedAddFile(inodeId, ..., addCloseOp.clientName,
                                addCloseOp.clientMachine, addCloseOp.storagePolicyId)
>++ fsNamesys.leaseManager.addLease(addCloseOp.clientName, path)
>++ // add the op into retry cache if necessary
>++ if (toAddRetryCache) {
>+++ HdfsFileStatus stat =
                                fsNamesys.dir.createFileStatus(HdfsFileStatus.EMPTY_NAME, newFile, ...)
>++++ fsNamesys.addCacheEntryWithPayload(addCloseOp.rpcClientId,
                                addCloseOp.rpcCallId, stat)
>++ }
>+ } else { // This is OP_ADD on an existing file
>++ if (!oldFile.isUnderConstruction()) {
>+++ // This is case 3: a call to append() on an already-closed file.
>++++ LocatedBlock lb = fsNamesys.prepareFileForWrite(path, oldFile,
                                addCloseOp.clientName, addCloseOp.clientMachine,
                                false, iip.getLatestSnapshotId(), false)
>++++ newFile = INodeFile.valueOf(fsDir.getInode(path), path, true)
>++++ // add the op into retry cache is necessary
>++++ if (toAddRetryCache) {
>+++++ fsNamesys.addCacheEntryWithPayload(addCloseOp.rpcClientId, addCloseOp.rpcCallId, lb)
>+++++ }
>++ }
>+ }
>+ // Fall-through for case 2.
>+ // Regardless of whether it's a new file or an updated file, update the block list.
>+ newFile.setAccessTime(addCloseOp.atime, Snapshot.CURRENT_STATE_ID)
>+ newFile.setModificationTime(addCloseOp.mtime, Snapshot.CURRENT_STATE_ID)
>+ updateBlocks(fsDir, addCloseOp, newFile)
> case OP_CLOSE: {
>+ ...

```



```

> case OP_UPDATE_BLOCKS:
>+ UpdateBlocksOp updateOp = (UpdateBlocksOp) op
>+ String path = renameReservedPathsOnUpgrade(updateOp.path, logVersion)
>+ INodeFile oldFile = INodeFile.valueOf(fsDir.getInode(path), path)
>+ updateBlocks(fsDir, updateOp, oldFile) // Update in-memory data structures
>+ if (toAddRetryCache) {
>++ fsNamesys.addCacheEntry(updateOp.rpcClientId, updateOp.rpcCallId)
>+ }
> case OP_ADD_BLOCK:
>+ AddBlockOp addBlockOp = (AddBlockOp) op
>+ path = renameReservedPathsOnUpgrade(addBlockOp.getPath(), logVersion)
>+ INodeFile oldFile = INodeFile.valueOf(fsDir.getInode(path), path)
>+ addNewBlock(fsDir, addBlockOp, oldFile) // add the new block to the INodeFile
> case ...
> } //end switch

```

注意,所谓对 FSImage 重演主要是对目录的重演,只是把目录修改得更符合已经存在于众多 DataNode 上的客观存在,而对于 DataNode 上的那些数据块复份,则不存在重演的问题,对数据块复份只有增补和删除的问题。

以数据块更改的重演为例,就是使目录中对于数据块的记载(比方说最后一块的内容长度)与实际情况相符。

```

[FSImage.loadFSImage() > loadEdits() > FSEditLogLoader.loadFSEdits()
> loadEditRecords() > applyEditLogOp() > updateBlocks()]

```

```

FSEditLogLoader.updateBlocks(FSDirectory fsDir, BlockListUpdatingOp op, INodeFile file)
> BlockInfo[] oldBlocks = file.getBlocks() //这里的 file 是个代表着文件的 INode
> Block[] newBlocks = op.getBlocks() //这组 Block 是操作 OP_UPDATE_BLOCKS 的目标
> String path = op.getPath()
> // Are we only updating the last block's gen stamp.
> boolean isGenStampUpdate = oldBlocks.length == newBlocks.length
> // First, update blocks in common
> for (int i = 0; i < oldBlocks.length && i < newBlocks.length; i++) {
>+ BlockInfo oldBlock = oldBlocks[i]
>+ Block newBlock = newBlocks[i]
>+ boolean isLastBlock = i == newBlocks.length - 1
>+ if (oldBlock.getBlockId() != newBlock.getBlockId() ||
    (oldBlock.getGenerationStamp() != newBlock.getGenerationStamp()
      && !(isGenStampUpdate && isLastBlock))) {
>++ IOException("Mismatched block IDs or generation stamps, " + ...)
>+ }

```

```

>+ oldBlock.setNumBytes(newBlock.getNumBytes())
>+ boolean changeMade = oldBlock.getGenerationStamp() != newBlock.getGenerationStamp()
>+ oldBlock.setGenerationStamp(newBlock.getGenerationStamp())
>+ if (oldBlock instanceof BlockInfoUnderConstruction &&
                                (!isLastBlock || op.shouldCompleteLastBlock())) {
>++ changeMade = true
>++ fsNamesys.getBlockManager().forceCompleteBlock(file,
                                (BlockInfoUnderConstruction) oldBlock)
>+ }
>+ if (changeMade) {    //如果有了变化
>++ // The state or gen - stamp of the block has changed. So, we may be able to process
>++ // some messages from datanodes that we previously were unable to process.
>++ fsNamesys.getBlockManager().processQueuedMessagesForBlock(newBlock)
>+ }    //处理有关该 Block 的其他有待进行的操作
> } //end for
> if (newBlocks.length < oldBlocks.length) {    //这里在删除 Block(只能是最后一块)
>+ // We're removing a block from the file, e.g. abandonBlock(...)
>+ if (!file.isUnderConstruction()) {    //只有尚在构筑,还没有关闭的文件才允许
>++ IOException("Trying to remove a block from file " + path +
                                "which is not under construction.")
>+ }
>+ if (newBlocks.length != oldBlocks.length - 1) {    //一次只能删除一个 Block
>++ IOException("Trying to remove more than one block from file " + path)
>+ }
>+ Block oldBlock = oldBlocks[oldBlocks.length - 1]
>+ boolean removed = fsDir.unprotectedRemoveBlock(path, file, oldBlock)
                                //在本级目录中从该文件的 INode 中删去这个 Block
>+ if (!removed && !(op instanceof UpdateBlocksOp)) {
>++ IOException("Trying to delete non - existant block " + oldBlock)
>+ }
> } else if (newBlocks.length > oldBlocks.length) { // We're adding blocks,这是在增添 Block
>+ for (int i = oldBlocks.length; i < newBlocks.length; i++) {
>++ Block newBlock = newBlocks[i]
>++ if (!op.shouldCompleteLastBlock()) {
>+++ BlockInfo newBI = new BlockInfoUnderConstruction(newBlock,
                                file.getBlockReplication())
>++ } else {
>+++ newBI = new BlockInfo(newBlock, file.getBlockReplication())
>++ }
>++ fsNamesys.getBlockManager().addBlockCollection(newBI, file)

```

```

//将此 Block 添加到 BlocksMap 中
> ++ file.addBlock(newBI) //将此 Block 添加到文件的 INode 中

> ++ fsNamesys.getBlockManager().processQueuedMessagesForBlock(newBlock)
//处理有关该 Block 的其他有待进行的操作

> + } //end for
> } //end if else if

```

摘要中已经加了一些注释,留给读者自己结合源代码进一步深入理解。

11.9 版本升级与故障恢复

在运行中,有时候会遇上版本升级(RollingUpgrade)和故障恢复(Recover)的问题。

所谓版本升级,并不是指因文件系统内容的变化而导致的 FSImage 改变,事实上 FSImage 随时都在改变,它的硬拷贝即 checkpoint 也是每做一次就必然会有不同。版本升级是指因为软件版本升级而带来的变化。比方说,假定文件中最后一个 Block 的复份原来都放在 tmp 目录下,但是软件升级了,新版软件决定要增加一个 rbw 目录,要将最后一个 Block 的复份改放在 rbw 目录下。这样一来,软件升级后再开机的时候那些 DataNode 就要在宿主文件系统中创建目录 rbw,并把原先在 tmp 目录下的那些文件转移到 rbw 目录下。这样的情况,DataNode 上会有,NameNode 上也会有,而且也许更多。甚至连文件系统映像文件的格式,也不能说没有改变的可能。

对于版本升级所带来问题的应对,是只局限于特定版本的,因为每次版本升级改变些什么只有这个具体的新版本才知道,所以只要我们拿到一个新版本的 Hadoop,它的 HDFS 子系统的程序代码中可能就蕴含着对于版本升级的处理,版本与版本之间各不相同。

进一步,版本升级以后还可能会有因为对新版本觉得不满意而要“回滚(RollBack)”到某个老版本或“降级(Downgrade)”的问题。

而故障恢复,则是比方说系统或个别节点在运行中突然断电,或者机器“死了”再重新开机时如何恢复到故障之前状态的问题。

这些问题都发生在重新开机的时候。

受版本升级影响最大的是 NameNode,发生在开机后装载文件系统时。

```
[NameNode.initialize()>NameNode.loadNamesystem()]
```

```
NameNode.loadNamesystem()
```

```

> this.namesystem = FSNamesystem.loadFromDisk(conf) //从磁盘加载
>> nd = FSNamesystem.getNamespaceDirs(conf)
>> ed = FSNamesystem.getNamespaceEditsDirs(conf)
>> FSImage fsImage = new FSImage(conf, nd, ed)
>> namesystem = new FSNamesystem(conf, fsImage, false)
>> StartupOption startOpt = NameNode.getStartupOption(conf)
>> if (startOpt == StartupOption.RECOVER) //如果命令行中给定了 recover 选项

```

```

>>>+ namesystem.setSafeMode(SafeModeAction.SAFEMODE_ENTER)
//恢复须在安全模式下进行
>>> namesystem.loadFSImage(startOpt) == FSNamesystem.loadFSImage(startOpt)
>>>> FSImage fsImage = getFSImage()
>>>>> return fsImage
>>>> if (startOpt == StartupOption.FORMAT) { //如果命令行中给定了 format 选项
>>>>+ fsImage.format(this, fsImage.getStorage().determineClusterId()); // reuse current id
>>>>+ startOpt = StartupOption.REGULAR
>>>> }
>>>> MetaRecoveryContext recovery = startOpt.createRecoveryContext()
>>>> boolean staleImage = fsImage.recoverTransitionRead(startOpt, this, recovery)
>>>> if (RollingUpgradeStartupOption.ROLLBACK.matches(startOpt) ||
RollingUpgradeStartupOption.DOWNGRADE.matches(startOpt)) {
//如果命令行中给定了回滚或降级选项
>>>>+ rollingUpgradeInfo = null //既然是 RollBack,当然就不会有 RollingUpgrade
>>>> }
>>>> boolean needToSave = staleImage &&!haEnabled &&!isRollingUpgrade()
>>>> if (needToSave) { //如果需要保存 Namespace
>>>>+ fsImage.saveNamespace(this)
>>>> } else { //无需保存
>>>>+ updateStorageVersionForRollingUpgrade(fsImage.getLayoutVersion(), startOpt)
>>>>+ prog.beginPhase(Phase.SAVING_CHECKPOINT)
>>>>+ prog.endPhase(Phase.SAVING_CHECKPOINT)
>>>> }
>>>> if (!haEnabled || (haEnabled && startOpt == StartupOption.UPGRADE)
|| (haEnabled && startOpt == StartupOption.UPGRADEONLY)) {
>>>>+ fsImage.openEditLogForWrite() //打开日志准备写入
>>>> }
>>>> imageLoadComplete()
>>>>> setImageLoaded()

```

NameNode.loadNamesystem()中唯一的操作就是 loadFromDisk(),即从磁盘加载文件系统;而后的核心则是 loadFSImage()。注意,在调用 loadFSImage()时传下的参数是 startOpt,这是启动 NameNode 时的命令行选项,这些选项指定了是否需要恢复、回滚、降级。按理说升级是不需要在命令行中指定的,但这里也有特殊的考虑:“-upgradeOnly”表示光升一下级,升完了就关机;“-upgrade”则表示升级并运行,后面还可以跟一个选项“-clusterid cid”;二者都不用就表示暂不升级,仍按原先的版本运行(这意味着新版本须能兼容老版本)。

然后,在 loadFSImage()里面,第一个大动作是 recoverTransitionRead(),我们这就来看看 recoverTransitionRead()(注意上一层的参数 startOpt 继续传了下来):

```
[NameNode.initialize() > NameNode.loadNamesystem() > FSNamesystem.loadFromDisk()]
```

```

> FSNamesystem.loadFSImage() > FSImage.recoverTransitionRead()]

FSImage.recoverTransitionRead(StartupOption startOpt,
                               FSNamesystem target, MetaRecoveryContext recovery)
> Collection<URI> imageDirs = storage.getImageDirectories()
> Collection<URI> editsDirs = editLog.getEditURIs()
> // 1. For each data directory calculate its state and check whether
                               all is consistent before transitioning.
> dataDirStates = new HashMap<StorageDirectory, StorageState>()
> boolean isFormatted = recoverStorageDirs(startOpt, dataDirStates)
> if (!isFormatted && startOpt != StartupOption.ROLLBACK &&
                               startOpt != StartupOption.IMPORT) {
>+ IOException("NameNode is not formatted.")
> }
> layoutVersion = storage.getLayoutVersion()
> if (startOpt == StartupOption.METADATAVERSION) {
>+ System.out.println("HDFS Image Version: " + layoutVersion)
>+ System.out.println("Software format version: " +
                               HdfsConstants.NAMENODE_LAYOUT_VERSION)
>+ return false
> }
> if (layoutVersion < Storage.LAST_PRE_UPGRADE_LAYOUT_VERSION) {
>+ NNStorage.checkVersionUpgradable(storage.getLayoutVersion())//检查并级前后的版本
> }
> if (startOpt != StartupOption.UPGRADE &&
                               startOpt != StartupOption.UPGRADEONLY &&
                               !RollingUpgradeStartupOption.STARTED.matches(startOpt) &&
                               layoutVersion < Storage.LAST_PRE_UPGRADE_LAYOUT_VERSION &&
                               layoutVersion != HdfsConstants.NAMENODE_LAYOUT_VERSION) {
>+ IOException("\nFile system image contains an old layout version " ...)
> }
> storage.processStartupOptionsForUpgrade(startOpt, layoutVersion)
                               //处理与升级选项相关的参数,例如 ClusterID 和 BlockPoolID
> // 2. Format unformatted dirs.
> for (Iterator<StorageDirectory> it = storage.dirIterator(); it.hasNext();) {
>+ StorageDirectory sd = it.next()
>+ StorageState curState = dataDirStates.get(sd)
>+ switch(curState) {
>+ case NON_EXISTENT;
>+ + IOException(StorageState.NON_EXISTENT + " state cannot be here")

```

```

>+ case NOT_FORMATTED;
>++ LOG.info("Storage directory " + sd.getRoot() + " is not formatted.")
>++ LOG.info("Formatting ...")
>++ sd.clearDirectory(); // create empty current dir
>+ }
> } //end for
> // 3. Do transitions
> switch(startOpt) {
> case UPGRADE: case UPGRADEONLY: //需要升级
>+ doUpgrade(target)
>+ return false; // upgrade saved image already
> case IMPORT: //导入一个现成的 FSImage 映像,即 Checkpoint
>+ doImportCheckpoint(target)
>+ return false; // import checkpoint saved image already
> case ROLLBACK: //需要回滚,须用专用的 rollback 命令
>+ throw new AssertionError("Rollback is now a standalone command,
                                NameNode should not be starting with this option.")
> case REGULAR:
> default: // just load the image
> } //end switch
> return loadFSImage(target, startOpt, recovery)
    == FSImage.loadFSImage(target, startOpt, recovery)

```

注意,这里的前两个 case 在处理完 doUpgrade() 或 doImportCheckpoint() 以后就直接返回了。此外,如果命令行中用了 rollback 选项,那么这里会发生异常,说 rollback 需要用独立的命令,从而也跳出了 recoverTransitionRead() 的执行。只有在既不要求版本升级,也不要求故障恢复或回滚的情况下才会执行后面的 loadFSImage()。

如果需要升级,就由 FSImage.doUpgrade() 实施版本升级。注意,具体的升级措施是与具体的版本紧密相连的,所以这个版本的 FSImage.doUpgrade() 与那个版本的 FSImage.doUpgrade() 可能完全不一样。下面的摘要只是让读者感受一下:

```

[NameNode.loadNamesystem() > FSNamesystem.loadFromDisk() >
FSNamesystem.loadFSImage() > FSImage.recoverTransitionRead() > FSImage.doUpgrade()]

```

```

FSImage.doUpgrade(FSNamesystem target)
> checkUpgrade(target)
> // load the latest image
> this.loadFSImage(target, StartupOption.UPGRADE, null)
> // Do upgrade for each directory
> target.checkRollingUpgrade("upgrade namenode")
> oldLV = storage.getLayoutVersion()

```



```

> storage.layoutVersion = HdfsConstants.NAMENODE_LAYOUT_VERSION
> LOG.info("Starting upgrade of local storage directories." ...)
> // Do upgrade for each directory
> for (Iterator<StorageDirectory> it = storage.dirIterator(false); it.hasNext();) {
>+ StorageDirectory sd = it.next();
>+ NNUUpgradeUtil.doPreUpgrade(sd)
> }
> if (target.isHaEnabled()) {
>+ editLog.doPreUpgradeOfSharedLog()
> }
> saveFSImageInAllDirs(target, editLog.getLastWrittenTxId())
> // upgrade shared edit storage first
> if (target.isHaEnabled()) {
>+ editLog.doUpgradeOfSharedLog()
> }
> for (Iterator<StorageDirectory> it = storage.dirIterator(false); it.hasNext();) {
>+ StorageDirectory sd = it.next()
>+ NNUUpgradeUtil.doUpgrade(sd, storage)
>+> LOG.info("Performing upgrade of storage directory " + sd.getRoot())
>+> // Write the version file, since saveFsImage only makes the fsimage_<txid>,
                                     and the directory is otherwise empty.
>+> storage.writeProperties(sd)
>+> File prevDir = sd.getPreviousDir()
>+> File tmpDir = sd.getPreviousTmp()
>+> Preconditions.checkState(!prevDir.exists(), "previous directory must not exist for upgrade.")
>+> Preconditions.checkState(tmpDir.exists(), "previous.tmp directory must exist for upgrade.")
>+> // rename tmp to previous
>+> NNStorage.rename(tmpDir, prevDir)
> }
> isUpgradeFinalized = false
> if (!storage.getRemovedStorageDirs().isEmpty()) {
>+ IOException("Upgrade failed in " ...)
> }

```

只有在无须版本升级或回滚,也不需要故障恢复的情况下,也就是 REGULAR 或 default 的情况下,才会如常装载 FSImage。不过,调用 FSImage.loadFSImage() 的并不只是 recoverTransitionRead() 这么一处,所以在这里面还是会分情形处理 ROLLBACK:

```

[NameNode.initialize() > NameNode.loadNamesystem() > FSNamesystem.loadFromDisk()
> FSNamesystem.loadFSImage() > FSImage.recoverTransitionRead()
> FSImage.loadFSImage()]

```

```

FSImage.loadFSImage(FSNamesystem target, StartupOption startOpt,
                    MetaRecoveryContext recovery)

> boolean rollingRollback = RollingUpgradeStartupOption.ROLLBACK.matches(startOpt)
> if (rollingRollback) {
>+ // if it is rollback of rolling upgrade, only load from the rollback image
>+ EnumSet<NameNodeFile> nnfs = EnumSet.of(NameNodeFile.IMAGE_ROLLBACK)
> } else {
>+ // otherwise we can load from both IMAGE and IMAGE_ROLLBACK
>+ nnfs = EnumSet.of(NameNodeFile.IMAGE, NameNodeFile.IMAGE_ROLLBACK)
> }

> FSImageStorageInspector inspector = storage.readAndInspectDirs(nnfs, startOpt)
> isUpgradeFinalized = inspector.isUpgradeFinalized()
> List<FSImageFile> imageFiles = inspector.getLatestImages()
> prog.beginPhase(Phase.LOADING_FSIMAGE)
> File phaseFile = imageFiles.get(0).getFile()
> prog.setFile(Phase.LOADING_FSIMAGE, phaseFile.getAbsolutePath())
> prog.setSize(Phase.LOADING_FSIMAGE, phaseFile.length())
> boolean needToSave = inspector.needToSave()
> initEditLog(startOpt)
> if (NameNodeLayoutVersion.supports(
    LayoutVersion.Feature.TXID_BASED_LAYOUT, getLayoutVersion())) {
>+ toAtLeastTxId = editLog.isOpenForWrite()?inspector.getMaxSeenTxId() : 0
>+ if (rollingRollback) {
>++ toAtLeastTxId = imageFiles.get(0).getCheckpointTxId() + 2
>+ }
>+ editStreams = editLog.selectInputStreams(
    imageFiles.get(0).getCheckpointTxId() + 1, toAtLeastTxId, recovery, false)
> } else {
>+ editStreams = FSImagePreTransactionalStorageInspector.getEditLogStreams(storage)
> }

> maxOpSize = conf.getInt(DFSConfigKeys.DFS_NAMENODE_MAX_OP_SIZE_KEY, ...)
> for (EditLogInputStream elis : editStreams) elis.setMaxOpSize(maxOpSize)
> for (int i = 0; i < imageFiles.size(); i++) {
>+ imageFile = imageFiles.get(i)
>+ loadFSImageFile(target, recovery, imageFile, startOpt)
> }

> prog.endPhase(Phase.LOADING_FSIMAGE)
> if (!rollingRollback) { //如果不要求回滚
>+ txnsAdvanced = loadEdits(editStreams, target, startOpt, recovery)
>+ needToSave |= needsResaveBasedOnStaleCheckpoint(imageFile.getFile(), txnsAdvanced)

```

```

>+ if (RollingUpgradeStartupOption.DOWNGRADE.matches(startOpt)) {
>++ // rename rollback image if it is downgrade
>++ renameCheckpoint(NameNodeFile.IMAGE_ROLLBACK, NameNodeFile.IMAGE)
>+ }
> } else { //要求回滚
>+ // Trigger the rollback for rolling upgrade. Here lastAppliedTxId
                                equals to the last txid in rollback fsimage.
>+ rollingRollback(lastAppliedTxId + 1, imageFiles.get(0).getCheckpointTxId())
>+ needToSave = false
> }
> editLog.setNextTxId(lastAppliedTxId + 1)
> return needToSave

```

装载 FSImage 意味着读入保存在磁盘上的映像文件,这里又有版本的问题。

```

[FSNamesystem.loadFSImage()>FSImage.recoverTransitionRead()>FSImage.loadFSImage()
> loadFSImageFile()]

```

```

FSImage.loadFSImageFile(FSNamesystem target, MetaRecoveryContext recovery,
                                FSImageFile imageFile, StartupOption startupOption)
> LOG.debug("Planning to load image :\n" + imageFile)
> StorageDirectory sdForProperties = imageFile.sd
> storage.readProperties(sdForProperties, startupOption)
> if (NameNodeLayoutVersion.supports(
    LayoutVersion.Feature.TXID_BASED_LAYOUT, getLayoutVersion())) {
>+ // For txid-based layout, we should have a .md5 file next to the image file
>+ loadFSImage(imageFile.getFile(), target, recovery, isRollingRollback) //并非递归调用
> } else if (NameNodeLayoutVersion.supports(
    LayoutVersion.Feature.FSIMAGE_CHECKSUM, getLayoutVersion())) {
>+ // In 0.22, we have the checksum stored in the VERSION file.
>+ String md5 = storage.getDeprecatedProperty(
    NNStorage.DEPRECATED_MESSAGE_DIGEST_PROPERTY)
>+ loadFSImage(imageFile.getFile(), new MD5Hash(md5), target, recovery, false)
> } else { // We don't have any record of the md5sum
>+ loadFSImage(imageFile.getFile(), null, target, recovery, false)
>+> // BlockPoolId is required when the FsImageLoader loads the
                                rolling upgrade information. Make sure the ID is properly set.
>+> target.setBlockPoolId(this.getBlockPoolID())
>+> FSImageFormat.LoaderDelegator loader = FSImageFormat.newLoader(conf, target)
>+> loader.load(curFile, requireSameLayoutVersion)
>+> MD5Hash readImageMd5 = loader.getLoadedImageMd5()

```

```

>+> if (expectedMd5 != null && !expectedMd5.equals(readImageMd5)) {
>+>+ IOException("Image file " + curFile + " is corrupt with MD5 checksum of " ...)
>+> } //end else
>+> long txId = loader.getLoadedImageTxId()
>+> LOG.info("Loaded image for txid " + txId + " from " + curFile)
>+> lastAppliedTxId = txId
>+> storage.setMostRecentCheckpointInfo(txId, curFile.lastModified())
> }

```

注意,这里对 `loadFSImage()` 的调用并非递归调用。前面在 `recoverTransitionRead()` 中调用的是三个参数的 `loadFSImage()`,而这里调用的都不是这带三个参数的 `loadFSImage()`。

这里分三种情况以不同的参数序列调用 `loadFSImage()`,但是第一个参数都是一样的,都是 `imageFile.getFile()`,那就是所欲加载的映像文件,但是其他的参数就有不同了。比方说用不用 MD5 校验,那就是版本之间不同。所以,装载文件系统映像的时候会根据其 `LayoutVersion`,即映像格局的版本不同而有不同的处理。

再看升级后的回滚。如果对升级后的表现不满,觉得还不如升级之前好而决定要回滚,那就先关机(指 JVM),在再次开机(即启动 JVM)的命令行中加上要回滚的选项,这样 `NameNode` 起来之后就会先处理版本回滚。

```
[NameNode.createNameNode() >case ROLLBACK >NameNode.doRollback()]
```

NameNode.doRollback()

```

> String nsId = DFSUtil.getNamenodeNameServiceId(conf)
> String namenodeId = HAUtil.getNameNodeId(conf, nsId)
> initializeGenericKeys(conf, nsId, namenodeId)
> FSNamesystem nsys = new FSNamesystem(conf, new FSImage(conf))
> System.err.print("\nrollBack\n" will remove the current state of the file system,\n
    returning you to the state prior to initiating your recent.\n
    upgrade. This action is permanent and cannot be undone. If you\n
    are performing a rollback in an HA environment, you should be\n
    certain that no NameNode process is running on any host.")
> nsys.getFSImage().doRollback(nsys) == FSImage.doRollback(FSNamesystem fsns)
>> // Rollback is allowed only if there is a previous fs states in at least one of the storage
    directories. Directories that don't have previous state do not rollback
>> boolean canRollback = false
>> FSImage prevState = new FSImage(conf)
>> prevState.getStorage().layoutVersion = HdfsConstants.NAMENODE_LAYOUT_VERSION
>> for (Iterator<StorageDirectory> it = storage.dirIterator(false); it.hasNext();) {
>>+ StorageDirectory sd = it.next()
>>+ if (!NNUpgradeUtil.canRollBack(sd, storage, prevState.getStorage(),
    HdfsConstants.NAMENODE_LAYOUT_VERSION)) {

```

```

>> ++ continue
>> + }
>> + LOG.info("Can perform rollback for " + sd)
>> + canRollback = true
>> } //end for
>> if (fsns.isHaEnabled()) {
>> + // If HA is enabled, check if the shared log can be rolled back as well.
>> + editLog.initJournalsForWrite()
>> + boolean canRollBackSharedEditLog =
            editLog.canRollBackSharedLog(prevState.getStorage(),
            HdfsConstants.NAMENODE_LAYOUT_VERSION)
>> + if (canRollBackSharedEditLog) {
>> ++ LOG.info("Can perform rollback for shared edit log.")
>> ++ canRollback = true
>> + }
>> } //end if(fsns.isHaEnabled())
>> if (!canRollback) IOException(
    "Cannot rollback. None of the storage directories contain previous fs state.")
>> // Now that we know all directories are going to be consistent Do rollback for
            each directory containing previous state
>> for (Iterator<StorageDirectory> it = storage.dirIterator(false); it.hasNext();) {
            //在每个存储设备上实施回滚
>> + StorageDirectory sd = it.next()
>> + LOG.info("Rolling back storage directory " ...)
>> + NNUpgradeUtil.doRollBack(sd)
>> } //end if(fsns.isHaEnabled())
>> if (fsns.isHaEnabled()) { // If HA is enabled, try to roll back the shared log as well.
>> + editLog.doRollback() //若有共享的 EditLog 也需要回滚
>> } //end for
>> isUpgradeFinalized = true
>> prevState.close()

```

回滚并非在任何情况下都可进行,只有在一定的条件下才可以回滚,在有些情况下就回不去了。至于具体的回滚操作,一来篇幅所限,二来过于琐碎也不属于本书的主旋律,这里就不深入进去了。

除升级和回滚之外,恢复也是个常见的问题。NameNode 也是一上来就先处理恢复的问题:

```
[NameNode.createNameNode() > NameNode.doRecovery()]
```

NameNode.doRecovery()

```
> String nsId = DFSUtil.getNamenodeNameServiceId(conf)
```

```

> String namenodeId = HAUtil.getNameNodeId(conf, nsId)
> initializeGenericKeys(conf, nsId, namenodeId)
> if (startOpt.getForce() < MetaRecoveryContext.FORCE_ALL) {
>+ confirmPrompt("You have selected Metadata Recovery mode.
                This mode is intended to recover lost metadata on a corrupt filesystem.
                Metadata recovery mode often permanently deletes data from your
                HDFS filesystem. Please back up your edit log and fsimage before
                trying this!" + "Are you ready to proceed?(Y/N)\n"))
>+ if (!confirmPrompt("...")) {
>++ System.err.println("Recovery aborted at user request.\n")
>++ return //如果用户选择不往下继续,就在此返回了
>+ }
> }
> MetaRecoveryContext.LOG.info("starting recovery...") //开始恢复
> UserGroupInformation.setConfiguration(conf)
> FSNamesystem fsn = FSNamesystem.loadFromDisk(conf) //这个过程就是恢复的过程
> fsn.getFSImage().saveNamespace(fsn) //保存恢复以后的 Namespace
> MetaRecoveryContext.LOG.info("RECOVERY COMPLETE")

```

相比之下,故障恢复比较简单一些,loadFromDisk()的过程也就是恢复的过程。如前所述,从磁盘加载文件系统映像的过程包含了在所装载映像上逐条重演 EditLog 记录的过程,所以实际上对 loadFromDisk()的调用就是恢复文件系统的过程。

以上所说都是 NameNode 上的版本升级、回滚和故障恢复。DataNode 上也有版本升级和回滚的问题。当一个 DataNode 在集群中通过网络连接上一个 NameNode 时(在“联邦”模式下一个集群中可以有多多个 Namespace,从而有多多个 NameNode),就要在本地为这个 NameNode 创建一个 BlockPool,实际上是一个 BlockPoolSlice,用来为这个 NameNode 提供数据块复份的存储,这时候就会调用 DataNode.initBlockPool()。对于这个特定的 NameNode 而言,具体 DataNode 上的 BlockPool 就相当于它的一个虚拟存储设备。但是虚拟的 BlockPool 必须要落实到具体的存储设备上,这就是 DataStorage,实际上就是由该 DataNode 所在节点上宿主文件系统提供的存储空间。就像 NameNode 上有 NNStorage 一样,DataNode 上也有 DataStorage。就具体的 DataNode 而言,其 DataStorage 很可能不是新建的,而是原来就已存在,原来就有内容,于是就也有了版本升级和回滚的问题。不过 DataNode 上的版本升级和回滚不一定都是在开机的时候,每当将一个 BlockPoolSlice 或 Storage 加入到一个 DataNode 时就需要考虑升级或回滚。

这里的 BlockPoolSlice 代表着整个 BlockPool 落实在一个具体 DataNode 上的那一部分(称为 Slice),对此后面还要详述。而 Storage 则是具体 DataNode 上的虚拟“存储设备”,通常是宿主文件系统中的几个或几个目录。DataNode 在对具体 BlockPool 的初始化过程中辗转调用一个函数 BlockPoolSliceStorage.doTransition(),在这里决定是否需要升级或回滚:

```

★ [DataNode.initBlockPool()>DataNode.initStorage()>DataStorage.recoverTransitionRead()
  >BlockPoolSliceStorage.recoverTransitionRead()>BlockPoolSliceStorage.doTransition()]

```



```

BlockPoolSliceStorage.doTransition(DataNode datanode, StorageDirectory sd,
                                   NamespaceInfo nsInfo, StartupOption startOpt)
> if (startOpt == StartupOption.ROLLBACK && sd.getPreviousDir().exists()) {
>+ Preconditions.checkState(!getTrashRootDir(sd).exists(),
    sd.getPreviousDir() + " and " + getTrashRootDir(sd) + " should not " +
    " both be present.")
>+ doRollback(sd, nsInfo); // rollback if applicable,回滚
> } else {
>+ int restored = restoreBlockFilesFromTrash(getTrashRootDir(sd))
>+ LOG.info("Restored " + restored + " block files from trash.")
> }
> readProperties(sd)
> checkVersionUpgradable(this.layoutVersion)
> assert this.layoutVersion >= HdfsConstants.DATANODE_LAYOUT_VERSION
    : "Future version is not allowed"
> if (this.layoutVersion == HdfsConstants.DATANODE_LAYOUT_VERSION
    && this.cTime == nsInfo.getCTime()) {
>+ return; // regular startup,正常启动,无须回滚或升级
> }
> if (this.layoutVersion > HdfsConstants.DATANODE_LAYOUT_VERSION
    || this.cTime < nsInfo.getCTime()) {
>+ doUpgrade(datanode, sd, nsInfo); // upgrade,升级。
>+ return
> }
> throw new IOException("Datanode ...")

```

显然,首先是也许需要回滚,回滚以后一般也就跟正常的启动一样通过这里的 return 语句返回了。如果既非回滚,也非正常启动,那就应该是需要升级,否则就是异常了。

我们看一下版本升级。如前所述,这也仅对特定版本有意义,因为每个版本所做的改动会有不同:

```

[DataNode.initBlockPool()>DataNode.initStorage()>DataStorage.recoverTransitionRead()
> BlockPoolSliceStorage.recoverTransitionRead()>BlockPoolSliceStorage.doTransition()
> BlockPoolSliceStorage.doUpgrade()]

```

```

BlockPoolSliceStorage.doUpgrade(DataNode datanode, StorageDirectory bpSd, NamespaceInfo nsInfo)
> // Upgrading is applicable only to release with federation or after
> if (!DataNodeLayoutVersion.supports(LayoutVersion.Feature.FEDERATION, layoutVersion))
    return //只允许升级到支持“联邦”模式的版本,即比较新的版本
> LOG.info("Upgrading block pool storage directory " ...)

```

```

> // get <SD>/previous directory
> String dnRoot = getDataNodeStorageRoot(bpSd.getRoot().getCanonicalPath())
    //获取宿主文件系统为 DataNode 提供用作 Storage 的目录子树(根节点)
> StorageDirectory dnSdStorage = new StorageDirectory(new File(dnRoot))
    //为此子树创建一个 StorageDirectory 对象
> File dnPrevDir = dnSdStorage.getPreviousDir() //根节点下面的子目录 previous
> // If <SD>/previous directory exists delete it
> if (dnPrevDir.exists()) deleteDir(dnPrevDir)
> File bpCurDir = bpSd.getCurrentDir() //根节点下面具体 bpid 节点下的子目录 current
> File bpPrevDir = bpSd.getPreviousDir() //具体 bpid 节点下的子目录 previous
> assert bpCurDir.exists() : "BP level current directory must exist." //BP 是 BlockPool 的缩写
> cleanupDetachDir(new File(bpCurDir, DataStorage.STORAGE_DIR_DETACHED))
> // 1. Delete <SD>/current/<bpid>/previous dir before upgrading
> if (bpPrevDir.exists()) deleteDir(bpPrevDir) //删掉<SD>/current/<bpid>/previous 子目录
> File bpTmpDir = bpSd.getPreviousTmpDir()
> assert!bpTmpDir.exists() : "previous.tmp directory must not exist."
> // 2. Rename <SD>/current/<bpid>/current to <SD>/current/<bpid>/previous.tmp
> rename(bpCurDir, bpTmpDir)
> // 3. Create new <SD>/current with block files hardlinks and VERSION
> linkAllBlocks(datanode, bpTmpDir, bpCurDir)
> this.layoutVersion = HdfsConstants.DATANODE_LAYOUT_VERSION
> assert this.namespaceID == nsInfo.getNamespaceID()
    : "Data - node and name - node layout versions must be the same."
> writeProperties(bpSd)
> // 4.rename <SD>/current/<bpid>/previous.tmp to <SD>/current/<bpid>/previous
> rename(bpTmpDir, bpPrevDir)
> LOG.info("Upgrade of block pool " + blockpoolID + " at " + bpSd.getRoot() + " is complete")

```

可见,这一版本升级所处理的主要是目录结构的变化。

还有对于 DataStorage 的升级,那发生在为 DataNode 添加一个作为文件卷的 DataStorage 时,与此大同小异,我们就不看了。

再看回滚:

```
[BlockPoolSliceStorage.doTransition()> BlockPoolSliceStorage.doRollback()]
```

```

BlockPoolSliceStorage.doRollback() //Roll back to old snapshot at the block pool level
> File prevDir = bpSd.getPreviousDir()
> if (!prevDir.exists()) return //regular startup if previous dir does not exist
> // read attributes out of the VERSION file of previous directory
> BlockPoolSliceStorage prevInfo = new BlockPoolSliceStorage()
> prevInfo.readPreviousVersionProperties(bpSd)

```

```
> if (!(prevInfo.getLayoutVersion() >= HdfsConstants.DATANODE_LAYOUT_VERSION
    && prevInfo.getCTime() <= nsInfo.getCTime()))
>+ InconsistentFSStateException("Cannot rollback to a newer state" ...)//回滚都是有条件的
> LOG.info("Rolling back storage directory " ...)
> File tmpDir = bpSd.getRemovedTmp()
> assert!tmpDir.exists() : "removed.tmp directory must not exist."
> // 1. rename current to tmp
> File curDir = bpSd.getCurrentDir()
> assert curDir.exists() : "Current directory must exist."
> rename(curDir, tmpDir)
> // 2. rename previous to current
> rename(prevDir, curDir)
> // 3. delete removed.tmp dir
> deleteDir(tmpDir)
> LOG.info("Rollback of " + bpSd.getRoot() + " is complete")
```

同样,还有对于 `DataStorage` 的回滚,也是大同小异,我们也不看了。

第12章

HDFS 的 DataNode

12.1 DataNode

数据节点 DataNode 在 HDFS 文件系统中处于从属的地位,但是其结构却比处于主导地位的名称节点 NameNode 更复杂。这是因为:虽然 NameNode 起着目录的作用,但是文件的内容却是存储在 DataNode 上的,读写文件时一旦知道了哪一个块在什么节点上,或者指定存放在什么节点上,下面就不需要 NameNode 的介入了。而块的存取,却是颇为复杂的操作。再说 NameNode 是靠“听汇报”来掌握什么 DataNode 上存储着哪一些块的,这一来倒好像 DataNode 才是主动的,NameNode 反倒是被动的了。而且,新版 Hadoop 的设计还允许集群中有多个 NameNode,形成“联邦(Federal)”,一个 DataNode 可以为多个 NameNode 存储数据块,这一来 DataNode 就更加复杂了。

我们先看看 DataNode 的结构摘要,注意这里所说的数据块一般都是指数据块复份:

```
class DataNode extends Configured ...{} //DataNode 的结构部分
] BlockPoolManager blockPoolManager //同一文件卷的块属于相同 BlockPool
]] Map<String, BPOfferService> bpByNameserviceId //按 NameserviceId 查找
]] Map<String, BPOfferService> bpByBlockPoolId //按 BlockPoolId 查找
]] List<BPOfferService> offerServices //对于每个 Nameservice 都有个联络组
]]] List<BPServiceActor> bpServices //这个联络组中对于每个 NameNode 都有个联络员
] FsDatasetSpi<?extends FsVolumeSpi> data //数据块的集合,实际上是个 FsVolumeImpl
]] DataStorage dataStorage //代表着实际的存储
]] FsVolumeList volumes //数据块可能属于不同的文件卷
]] Map<String, DatanodeStorage> storageMap //按 storageUuid 查找
]] FsDatasetAsyncDiskService asyncDiskService //实现对磁盘的异步读写
]] Daemon lazyWriter //一个 LazyWriter 线程,仅在真有必要时才写磁盘
]] FsDatasetCache cacheManager //对于缓冲块的管理
]] ReplicaMap volumeMap //可以根据 BlockPoolId 和块号找到其复份的 ReplicaInfo
] DataStorage storage //代表着数据块的存储空间
] DataBlockScanner blockScanner //数据块扫描线程
] DirectoryScanner directoryScanner //目录扫描线程
] Daemon dataXceiverServer //数据收发线程,通过局域网收发数据
] ThreadGroup threadGroup //一组 dataXceiverServer 线程
```

```

] RPC.Server ipcServer           //提供 RPC 服务
] Daemon localDataXceiverServer //通过 Unix 域 socket 在本节点上收发数据的线程
] JvmPauseMonitor pauseMonitor  //监视 JVM 的运行是否停滞
] List<StorageLocation> dataDirs //本地宿主文件系统中用于 HDFS 块存储的目录
] Thread checkDiskErrorThread   //检查本地的磁盘操作是否有故障,线程

```

如上所述,从前一个 Hadoop 集群中只能有一个活跃的 NameNode,但是新版 Hadoop 对于数据块的存储已经支持所谓联邦模式,即允许在一个集群中可以有多个 NameNode,或者说多个文件名系统(FSNamesystem)。与此相应,在 DataNode 上就有多个 BlockPool,而 BlockPoolManager 就是这些 BlockPool 的管理者。一个 DataNode 只有一个 BlockPoolManager,是在 DataNode 的初始化阶段创建的。但是集群中可以有多个 FSNamesystem,所以 BlockPoolManager 中有两个映射表(Map)和一个 BPOfferService 的列表。一个 BPOfferService 相当于面向一个具体 FSNamesystem 的联络站,BP 就是 BlockPool 的缩写,所以这是“由 BlockPool 提供的服务”的意思。两个映射表的存在是为迅速找到具体的 BPOfferService,一个是按 NameserviceId 映射查找,另一个是按 BlockPoolId 映射查找。然后,进一步在每个 BPOfferService 内部又有一个 BPServiceActor 的列表,每个 BPServiceActor 相当于专责与某个 NameNode 联系的联络员。为什么针对同一个 FSNamesystem 还会有多个联络员呢?这是出于容错的考虑,因为同一个 FSNamesystem 也可能有作为热备份的 NameNode。

核心的部件当然是这里的数据,代表着数据块的集合,实现了对数据块集合的管理。从类型说明看,这是一个实现了 FsDatasetSpi 界面的某类对象,Spi 是“Service Provider Interface”的缩写,后面我们会看到这实际上是个 FsDatasetImpl 类对象。FsDatasetSpi 界面的定义是模板式的定义(泛型定义),可以针对不同类型的对象,而 FsDatasetImpl 所实现的 FsDatasetSpi 界面,则针对实现了 FsVolumeSpi 界面的类,实际上那是 FsVolumeImpl。之所以不确切说这就是 FsDatasetImpl,或 FsDatasetSpi<FsVolumeImpl>,是为了保持实现上的最大灵活性。

数据块最终总是存储在某种存储介质上,DataStorage 就代表着这样的存储介质、存储空间。不过 HDFS 所看到的存储介质并非物理的、实体的存储介质,例如磁盘,而是宿主机的文件系统,是宿主机文件系统上的若干目录。这些目录可以在磁盘上,也可以在 RAMDisk 上,还可以在别的介质上。DataStorage 类是对 Storage 类的扩充,我们在 NameNode 那一边看到的 NNStorage 也是对 Storage 类的扩充。所以,DataStorage 和 NNStorage 分别是 DataNode 和 NameNode 上的 Storage,二者都由宿主文件系统提供,只是使用方式不同。

一个 DataNode 上存储着一些什么块,并不是由 NameNode 告诉它的,而是反过来由 DataNode 自己扫描,得知本节点上有什么以后向 NameNode 报告的。而且这样的扫描和报告要周期地反复进行。DataNode 上有两个线程,即 DataBlockScanner 和 DirectoryScanner,就是专门从事这个工作的。

DataNode 上的节点间通信也比 NameNode 上复杂,因为 DataNode 不仅要面对 NameNode,还有 DataNode 互相之间的通信,特别是数据块的收发。DataNode 之间的通信是对等通信,没有固定的 Client 和 Server,每个 DataNode 都既是 Client 又是 Server,所以 DataNode 上专门有一组 dataXceiverServer 线程。当然,DataNode 上也要有 RPC 服务端,即 RPC.Server。

了解了 DataNode 的结构成分摘要,下面再看看它的操作方法摘要。

```
class DataNode extends Configured ...    /* DataNode 的操作方法部分 */
    implements InterDatanodeProtocol, ClientDatanodeProtocol, DataNodeMXBean{}
] DataNode(final Configuration conf,
    List<StorageLocation> dataDirs, SecureResources resources)    //构造方法
> super(conf)
> ...
> hostName = getHostName(conf)
> startDataNode(conf, dataDirs, resources)
>> ...
>> storage = new DataStorage()          //创建数据存储
>> registerMXBean()
>> initDataXceiver(conf)                //创建跨节点数据收发线程和本地数据收发线程
>>> tcpPeerServer = new TcpPeerServer(secureResources)
>>> this.threadGroup = new ThreadGroup("dataXceiverServer")
>>> this.dataXceiverServer = new Daemon(threadGroup,
    new DataXceiverServer(tcpPeerServer, conf, this)) //创建数据收发线程
>>> this.threadGroup.setDaemon(true); // auto destroy when empty
>>> domainPeerServer = getDomainPeerServer(conf, streamingAddr.getPort())
>>> this.localDataXceiverServer = new Daemon(threadGroup,
    new DataXceiverServer(domainPeerServer, conf, this)) //创建本地数据收发线程
>>> LOG.info("Listening on UNIX domain socket: " + domainPeerServer.getBindPath())
>>> this.shortCircuitRegistry = new ShortCircuitRegistry(conf)
>> startInfoServer(conf)                //创建并启动 Web 服务器 HttpServer2
>> pauseMonitor = new JvmPauseMonitor(conf) //创建 JVM 监视线程
>> pauseMonitor.start()
>> initIpcServer(conf)                  //创建节点间通信底层的服务端
>>> RPC.setProtocolEngine(conf, ClientDatanodeProtocolPB.class, ProtobufRpcEngine.class)
>>> clientDatanodeProtocolXlator =
    new ClientDatanodeProtocolServerSideTranslatorPB(this)
>>> BlockingService service =
    ClientDatanodeProtocolService.newReflectiveBlockingService(
        clientDatanodeProtocolXlator)
>>> ipcServer = new RPC.Builder(conf).setProtocol(ClientDatanodeProtocolPB.class)
    ...setSecretManager(blockPoolTokenSecretManager).build()
>>> interDatanodeProtocolXlator = new InterDatanodeProtocolServerSideTranslatorPB(this)
>>> service = InterDatanodeProtocolService.newReflectiveBlockingService(
    interDatanodeProtocolXlator)
>>> DFSUtil.addPBProtocol(conf, InterDatanodeProtocolPB.class, service, ipcServer)
>>> ipcServer.refreshServiceAcl(conf, new HDFSPolicyProvider())
```



```

>>> blockPoolManager = new BlockPoolManager(this) //创建 BlockPoolManager
>>> blockPoolManager.refreshNamenodes(conf) //建立与 NameNode 的联系
>>> readaheadPool = ReadaheadPool.getInstance()
] initBlockPool(BPOfferService bpos) //由 BPOfferService 在与 NameNode 建立联系后调用
> blockPoolManager.addBlockPool(bpos) //Register the new block pool with the BP manager
> initStorage(nsInfo)
>>> factory = FsDatasetSpi.Factory.getFactory(conf)
>>> ...
>>> data = factory.newInstance(this, storage, conf) //创建数据块集合 FsDatasetImpl
> checkDiskError()
> initPeriodicScanners(conf)
>>> initDataBlockScanner(conf)
>>>> blockScanner = new DataBlockScanner(this, data, conf) //创建数据块扫描线程
>>>> blockScanner.start() //启动其运行
>>> initDirectoryScanner(conf)
>>>> directoryScanner = new DirectoryScanner(data, conf) //创建目录扫描线程
>>>> directoryScanner.start() //启动其运行
> data.addBlockPool(nsInfo.getBlockPoolID(), conf)
] createBPRegistration(NamespaceInfo nsInfo) //由 BPOfferService 加以调用
> storageInfo = storage.getBPStorage(nsInfo.getBlockPoolID())
> dnId = new DatanodeID(...)
> new DatanodeRegistration(dnId, storageInfo, ...)
] connectToNN(InetSocketAddress nnAddr) //由具体的 BPServiceActor 加以调用
> new DatanodeProtocolClientSideTranslatorPB(nnAddr, conf)
] transferBlock(ExtendedBlock block, DatanodeInfo xferTargets[],
] class DataTransfer implements Runnable {} //不同 DataNode 之间的数据块传输线程
] getStorageLocations()
] class DataNodeDiskChecker{
] main(String args[])
> secureMain()
>>> datanode = createDataNode(args, null, resources)
>>>> dn = instantiateDataNode(args, conf, resources)
>>>>> conf = new HdfsConfiguration()
>>>>> dataLocations = getStorageLocations(conf)
//在配置文件 hdfs-default.xml 中寻找用于数据块文件存储的“dfs.datanode.data.dir”
//一般是 dfs/data,也可以不止一处
>>>>> UserGroupInformation.setConfiguration(conf)
>>>>> SecurityUtil.login()
>>>>> makeInstance(dataLocations, conf, resources)
>>>>>> LocalFileSystem localFS = FileSystem.getLocal(conf)

```

```
>>>>> List<StorageLocation> locations =
        checkStorageLocations(dataDirs, localFS, dataNodeDiskChecker)
>>>>> return new DataNode(conf, locations, resources) //创建 DataNode,执行 DataNode()
>>>> dn.runDataNodeDaemon()
>>>> blockPoolManager.startAll() //启动 BlockPoolManager
>>>> dataXceiverServer.start() //启动数据收发线程
>>>> localDataXceiverServer.start() //启动本地数据收发线程
>>>> ipcServer.start() //启动 ipc 服务线程
>>>> startPlugins(conf) //启动插件(如果有的话)
>>> if (datanode!= null) datanode.join()
```

可见创建 `DataNode` 对象的程序流程是 `main() > secureMain() > createDataNode() > instantiateDataNode() > makeInstance()`。当然,创建 `DataNode` 对象时会调用它的构造方法 `DataNode()`,于是就会在那里调用 `startDataNode()`,而 `DataStorage` 对象 `storage` 就是在 `startDataNode()` 里面创建的。

而 DataNode 的运行则是经 `createDataNode() > runDatanodeDaemon()` 启动的。所谓启动,实际上只是使 DataNode 的诸多部件各就各位,跟 NameNode 建立起通信渠道,做好了提供服务的准备而已。DataNode 从总体上说是被动的,除心跳 heartbeat 之外它不主动发起什么操作。所以,在没有用户即 Client 发起文件操作的情况下,整个 HDFS 文件系统中的活动就是由心跳所驱动,包括为提供心跳报告所需的准备,例如 DataBlockScanner 对数据块的扫描,以及执行 NameNode 接收心跳报告之后下达的命令等。

至于文件操作,虽然表面上可能是在 DataNode 所在节点上发起,但是发起者其实是应用,是用户,那是在不同的 Java 虚拟机上,逻辑上是在 HDFS 文件系统之外,只是这些应用可能运行在 DataNode 所在的节点上而已。

12.2 数据块的存储

前面简单介绍了 DataNode 的结构,现在进一步说明 HDFS 的数据块在集群中的 DataNode 上是怎样存储的。读者在看的时候最好也回顾一下前面有关 DataNode 上版本升级和回滚的代码摘要,那里就涉及宿主文件系统上的目录结构。

我在一台单机上安装了 Hadoop, 经过 format 操作进行格式化和创建 HDFS 文件系统, 并在上面运行了计算圆周率的那个示例, 然后用工具 du 观看宿主机文件系统的变化, 下面就是从 du 的输出结果中摘取出来的一些内容(没有用过 du 的读者可以在 Linux 上试用一下)。不过, 限于本书的页面宽度, 这里用省略号取代了节点 hdfs 前面的那部分路径, 那就是我指定安装 HDFS 的地方。另外, 由于是安装在单机上, 这个机器既是 DataNode, 又是 NameNode, 还又是帮着做 checkpoint 的 secondary NameNode, 所以宿主机的这个目录下既有 hdfs/dn, 又有 hdfs/nn, 还有 hdfs/snn, “snn”是“secondary name node”的缩写。本来这都应该是在不同的机器上的。

260 .../hdfs/dn/current/BP-2085968454-127.0.1.1-1426747700215/current/finalized
/subdir0/subdir0

```

264 .../hdfs/dn/current/BP-2085968454-127.0.1.1-1426747700215/current/finalized/subdir0
268 .../hdfs/dn/current/BP-2085968454-127.0.1.1-1426747700215/current/finalized
4 .../hdfs/dn/current/BP-2085968454-127.0.1.1-1426747700215/current/rbw
280 .../hdfs/dn/current/BP-2085968454-127.0.1.1-1426747700215/current
4 .../hdfs/dn/current/BP-2085968454-127.0.1.1-1426747700215/tmp
292 .../hdfs/dn/current/BP-2085968454-127.0.1.1-1426747700215
300 .../hdfs/dn/current
308 .../hdfs/dn
1048 .../hdfs/nn/current
1056 .../hdfs/nn
8 .../hdfs/snn
1376 .../hdfs

```

我们看到,在 hdfs/dn 下面有个子目录 current,这就是 DataNode 目前用来存储数据块的目录。因为这台机器还只运行了一小会儿,所以还没有别的子目录;如果运行的时间足够长,就会有别的子目录被创建出来,例如与 current 平行可以有 previous 等。这些后面还会讲到,现在我们就集中看这 current 子目录。

在 hdfs/dn/current 下面只有一个子目录 BP-2085968454-127.0.1.1-1426747700215,它代表着一个 BlockPool,BP 就是 BlockPool 的缩写。这说明本系统中只有一个 BlockPool,也就是只有一个 FSNamesystem,从而只有一个活跃的 NameNode。每个 BlockPool 都有个 BlockPoolId,BP-2085968454-127.0.1.1-1426747700215 就是这个 BlockPool 的 ID,其中有一些信息编码在里面。比较明显的是“127.0.1.1”,这是 NameNode 的 IP 地址,之所以是“127.0.1.1”,就是因为 NameNode 和 DataNode 同在一台机器上。此外,还有 DataNode 的 ID,也编码在里面。

在这个 BlockPool 下面,又有个子目录 current,但是这里还有个子目录 tmp。在这个 current 下面,则有 rbw 和 finalized。前者 rbw 是“Replica Being Written”的缩写,表示这个子目录中的块文件都是正在往里面写、尚未完成的复份(注意,在具体 DataNode 上的块文件都只是复份,一个块通常都有三个复份,存储在不同的节点上)。而 finalized 子目录下则是已经完成了的块文件。为什么块文件会有 rbw 和 finalized 之分呢? HDFS 的文件与普通的文件不同,它里面的内容一经“敲定”封存,即 finalized,就不能再改了,以后可以在文件的尾部添加,但是却不允许回过头去修改先前已经封存的内容。与普通的文件系统相比,HDFS 的这个特点显然是一种简化和限制,但这却是关键性的。要不然的话,HDFS 将会复杂很多,得要重新设计,效率也会下降很多。另一方面,对于大数据方面的应用,特别是数据挖掘,实际上也没有随机更改内容的要求。所以,一个块文件,在其内容尚未封存之前被放在 rbw 子目录中,一经封存就被转移到 finalized 子目录中。不过这样的安排,特别是有关具体的目录结构,也是逐步形成的,所以在版本升级和回滚的时候就可能涉及这些子目录的变动。

另外,在典型的情况下,一个 DataNode 上会有大量的块文件,所以在这个 finalized 子目录下还可以有个树形的结构,所以这里有/subdir0 和/subdir0/subdir0。

相比之下,在 NameNode 这一边,/hdfs/nn 下面也暂时还只有一个 current 子目录,这也是因为运行时间还很短,以后会创建与此平行的 previous 等子目录。

看了目录结构之后,我们再来看子目录中的文件。首先,我们可以看到在 `.../hdfs/dn/current` 和 `.../hdfs/nn/current` 这两个子目录下都有个文件叫 `VERSION`,并且在 `.../hdfs/dn/current/BP-2085968454-127.0.1.1-1426747700215/current` 下面也有 `VERSION`。顾名思义,这跟文件系统的版本有关,实际上还不只是有关版本的信息。

这三个文件的内容各不相同。先看 `.../hdfs/nn/current/VERSION`:

```
# Thu Mar 19 14:48:20 CST 2015
namespaceID = 964321229
clusterID = CID-958903fb-8406-4818-9c92-bce05283a412
cTime = 0
storageType = NAME_NODE
blockpoolID = BP-2085968454-127.0.1.1-1426747700215
layoutVersion = -60
```

如前所述,一个 `NameNode` 管理着一个 `FSNamesystem`,是该文件系统的查名服务节点。而 `FSNamesystem` 则是对“命名空间”即 `Namespace` 的实现,这是 HDFS 文件系统的上层。HDFS 的下层是数据块的 `BlockPool`。每个 `FSNamesystem` 有个 `Namespace` 的 `ID`,就是这里的 `namespaceID`。然后 `NameNode` 所在的集群也有个 `ID`,就是 `clusterID`。这里的 `storageType` 其实是指相关的信息存储在什么节点上,那就是 `NAME_NODE`。`FSNamesystem` 所管理的 HDFS 下层是 `BlockPool`,`Namespace` 中各个文件的内容以“块”为单位存储在某个分布于整个集群的 `BlockPool` 中,这里的 `blockpoolID` 就指明了具体的 `BlockPool`。最后,`layoutVersion` 是文件系统存储格局的版本号。一般,软件的版本是常常在升级的,但是 HDFS 文件系统在宿主文件系统上的存储格局(例如目录的设置)的变化就不会很频繁,存储格局的变化意味着重大的变更,所以这个版本号很重要。

再看 `.../hdfs/dn/current/VERSION`:

```
# Thu Mar 19 14:49:56 CST 2015
storageID = DS-00b190c5-ae5f-4458-97ef-9fe9b8cbf694
clusterID = CID-958903fb-8406-4818-9c92-bce05283a412
cTime = 0
datanodeUuid = 5916d90a-01c6-43cd-ae9a-d8dfb8d24af3
storageType = DATA_NODE
layoutVersion = -56
```

先看 `clusterID`,这与 `NameNode` 所在的 `clusterID` 相符,说明这个 `DataNode` 与 `NameNode` 在同一个集群中。但是具体的 `DataNode` 当然也需要有个 `ID`,这就是它的 `datanodeUuid`。`DataNode` 是存储数据块的,它的存储介质也得有个 `ID`,那就是这里的 `storageID`。最后,`storageType` 则是 `DATA_NODE`;而 `layoutVersion` 则与 `NameNode` 上的无关,这只是 `DataNode` 节点上的事。

然后是 `.../hdfs/dn/current/BP-2085968454-127.0.1.1-1426747700215/VERSION`:

```
# Thu Mar 19 14:49:56 CST 2015
namespaceID = 964321229
```

```
cTime = 0
blockpoolID = BP-2085968454-127.0.1.1-1426747700215
layoutVersion = -56
```

注意,这个 VERSION 文件所在的位置是在一个具体 BlockPool 的子目录下,所以它所描述的不是整个 DataNode 而是具体的 BlockPool。显然,这个 BlockPool 所对应的 Namespace 就是上面那个 NameNode 上的 namespace,二者的 NamespaceID 相同。而这里的这个 blockpoolID,当然就是编码在目录路径中的 BlockPool ID。

那么这些 VERSION 文件是干什么用的呢?我们可以设想,当 NameNode 或 DataNode 被启动运行时,它们能从配置文件知道 HDFS 文件系统在宿主文件系统即本地文件系统的位置,从而找到 hdfs 这个目录节点。然而再往下呢?NameNode 得要知道自己所对应、所管理的 BlockPool 是哪一个。DataNode 则要知道自己所支持的各个 BlockPool 分别对应着什么 Namespace。另一方面,十分重要的是,不管是 NameNode 还是 DataNode,其目录架构也并非一成不变,因而软件必须与之配套,所以一定得有个版本号,这应该就是文件名 VERSION 的由来。

总之,VERSION 文件中所提供的是关于 HDFS 文件存储格局的信息。显然这些信息不应只停留在磁盘文件中,而应该被读入内存的某个数据结构或者说某个“对象”之中(因为现在都是“数据结构+操作”了)。Hadoop 的代码中为此定义了一个 StorageInfo 类,下面是其摘要:

```
class StorageInfo {}
] int layoutVersion;           // layout version of the storage data
] int namespaceID;             // id of the file system
] String clusterID;            // id of the cluster
] long cTime                    // creation time of the file system state
] NodeType storageType;         // Type of the node using this storage
] static final String STORAGE_FILE_VERSION = "VERSION"
] StorageInfo(int layoutV, int nsID, String cid, long cT, NodeType type)
] versionSupportsFederation(Map<Integer, SortedSet<LayoutFeature>> map)
    > return LayoutVersion.supports(map, LayoutVersion.Feature.FEDERATION, layoutVersion)
] readPropertiesFile(File from) //从属性文件读入,VERSION就是个属性文件
```

将此类对象的数据部分(数据结构)与前面几个 VERSION 文件的内容进行比较,就可看出 StorageInfo 中的信息都来自前者,但是这里并不需要承载 VERSION 文件中的所有内容,这里没有 blockpoolID,也没有 storageID 和 datanodeUuid。

StorageInfo 并不代表一个存储设备或介质,而只是提供了一些基本信息。抽象类 Storage 是对 StorageInfo 的扩展,下面是它的摘要:

```
abstract class Storage extends StorageInfo {}
] List<StorageDirectory> storageDirs //一组存储目录,StorageDirectory的定义见下
] getFiles(StorageDirType dirType, String fileName)
] analyzeStorage(StartupOption startOpt, Storage storage)
//Check consistency of the storage directory
```

```

] doRecover(StorageState curState)
    //Complete or recover storage state from previously failed transition.
] class StorageDirectory implements FormatConfirmable {} //StorageDirectory 的定义
]] File root; // root directory
]] StorageDirType dirType; // storage dir type
]] String storageUuid // Storage directory identifier,storageID 就是来自这里。
]] analyzeStorage(StartupOption startOpt, Storage storage)
]] doRecover(StorageState curState)
]] getVersionFile()
    > return new File(new File(root, STORAGE_DIR_CURRENT), STORAGE_FILE_VERSION)
]] getPreviousVersionFile()
    > return new File(new File(root, STORAGE_DIR_PREVIOUS), STORAGE_FILE_VERSION)
]] getPreviousCheckpoint()
    > return new File(root, STORAGE_PREVIOUS_CKPT) // "previous.checkpoint"

```

这就是说,一个 Storage,即逻辑意义上的存储设备或存储介质,是由(宿主机上的)一个文件目录子树构成的,这就是“存储目录”(即 StorageDirectory)。不同的存储目录并不意味着所存储的一定都是不同的数据块,而可能只是代表着这个 DataNode 在不同的时间点上和不同情况下存储的数据块。比方说,经过一段时间的运行之后,DataNode 会在某个时间点上创建与 current 并行的 previous 子目录,并把当时 current 子目录下的数据块全都转移到 previous 下面,再把它们都连接到 current 下面。再以后,或许就有了 previous 和 current 两个版本,一旦有需要就可以通过回滚恢复到 previous 版本。

不过在 Storage 中仍然没有定义 blockpoolID,也没有定义 datanodeUuid。

Storage 是个抽象类,一定要经过扩充落实才可以实际存在。在 DataNode 上有两个类是对于 Storage 的扩充,一个是 BlockPoolSliceStorage,另一个是 DataStorage。二者都是对 Storage 的扩充,但处于不同的层次。前者只是代表着某个 BlockPool 分布在本 DataNode 上的那个“片”,即 Slice;后者则更为宏观,可以代表对于多个 BlockPool 的支持。

先看 BlockPoolSliceStorage 的摘要:

```

class BlockPoolSliceStorage extends Storage {}
] String blockpoolID = ""; // id of the blockpool,表示本片属于哪一个 BlockPool
] recoverTransitionRead(DataNode datanode, NamespaceInfo nsInfo,
    Collection<File> dataDirs, StartupOption startOpt)
] format(File dnCurDir, NamespaceInfo nsInfo)
    > File curBpDir = getBpRoot(nsInfo.getBlockPoolID(), dnCurDir)
    > StorageDirectory bpSdir = new StorageDirectory(curBpDir)
    > format(bpSdir, nsInfo)
] doTransition(DataNode datanode, StorageDirectory sd, NamespaceInfo nsInfo,
    StartupOption startOpt)
] doUpgrade(DataNode datanode, StorageDirectory bpSd, NamespaceInfo nsInfo)
] doRollback(StorageDirectory bpSd, NamespaceInfo nsInfo)

```



```
] doFinalize(File dnCurDir)
] linkAllBlocks(DataNode datanode, File fromDir, File toDir)
```

NameNode 所看到的全局性的 BlockPool, 具体到某个 DataNode 上就是一个 BlockPoolSlice。BlockPoolSlice 代表着由这个 NameNode 安排存储在某个具体 DataNode 上的所有数据块(复份)的集合。这些数据块总得要存储在一组物理的容器里, 而 BlockPoolSliceStorage 就代表着这组物理容器。注意, 这个类是对 Storage 的扩充, 所以就继承了 Storage 中的那个 storageDirs, 这就是用来支持一个具体 BlockPoolSlice 的那组物理容器。

而 DataStorage, 则是对抽象类 Storage 的扩充, 它代表着 DataNode 的某个磁盘上的全部物理容器, 因为在联邦模式下 DataNode 可以为不止一个的 NameNode 提供存储, 从而可以有不只一个的 BlockPoolSlice; 与此相应地在.../hdfs/dn/current 目录节点下可以有多个 BP 子目录, 分别用来存储这些 BlockPoolSlice 的数据块。不过, 即使可以有多个 BlockPoolSlice, 从而有多个 BP 子目录, 它们总还是在同一个磁盘介质上, 也就是宿主系统的同一个文件卷上。下面是 DataStorage 的摘要:

```
class DataStorage extends Storage {}
] List<StorageDirectory> storageDirs //这是在抽象类 Storage 中定义的
] String datanodeUuid //在哪个 DataNode 上
] Map<String, BlockPoolSliceStorage> bpStorageMap //一组 BlockPoolSliceStorage
] writeAll(Collection<StorageDirectory> dirs)
  > this.layoutVersion = getServiceLayoutVersion()
  > for (StorageDirectory dir : dirs) writeProperties(dir)
] addStorageLocations(DataNode datanode, NamespaceInfo nsInfo,
  Collection<StorageLocation> dataDirs, StartupOption startOpt)
] recoverTransitionRead(DataNode datanode, NamespaceInfo nsInfo,
  Collection<StorageLocation> dataDirs, StartupOption startOpt)
] makeBlockPoolDataDir(Collection<File> dataDirs, Configuration conf)
] format(StorageDirectory sd, NamespaceInfo nsInfo, String datanodeUuid)
] doTransition(DataNode datanode, StorageDirectory sd,
  NamespaceInfo nsInfo, StartupOption startOpt)
] doUpgrade(DataNode datanode, StorageDirectory sd, NamespaceInfo nsInfo)
] doRollback(StorageDirectory sd, NamespaceInfo nsInfo)
] doFinalize(StorageDirectory sd)
] addBlockPoolStorage(String bpID, BlockPoolSliceStorage bpStorage)
  > if (!this.bpStorageMap.containsKey(bpID)) {
  >+ this.bpStorageMap.put(bpID, bpStorage)
  > }
```

数据部分的 datanodeUuid 说明本存储设备在哪个 DataNode 上, bpStorageMap 则是一组 BlockPoolSliceStorage 的 Map, 以便根据 blockpoolID 找到相应的 BlockPoolSliceStorage 对象。DataStorage 也是对 Storage 的扩充, 所以也有个 storageDirs, storageDirs 的类型是 List<StorageDirectory>, 实际上是具体 DataNode 的某个文件卷上用于 HDFS 数据块存储

的所有子目录的集合,而 bpStorageMap 中的每个 BlockPoolSliceStorage,则又有它自己的 storageDirs,把所有 BlockPoolSliceStorage 的 storageDirs 合在一起,就是 DataStorage 的 storageDirs。注意 DataNode 的一个文件卷其实只是宿主文件系统中的子树。

DataStorage 类的定义中还有一些静态常数定义,这些常数定义告诉我们相关文件和子目录的命名规则。我们不妨从另一个视角对 DataStorage 类做个摘要:

```
class DataStorage {
    //DataStorage 类的一些静态常数定义
    ] static String BLOCK_SUBDIR_PREFIX = "subdir"
    ] static String BLOCK_FILE_PREFIX = "blk_"
    ] static String COPY_FILE_PREFIX = "dncp_"
    ] static String STORAGE_DIR_DETACHED = "detach"
    ] static String STORAGE_DIR_RBW = "rbw"
    ] static String STORAGE_DIR_FINALIZED = "finalized"
    ] static String STORAGE_DIR_LAZY_PERSIST = "lazypersist"
    ] static String STORAGE_DIR_TMP = "tmp"
```

这说明,用来存储数据块的子目录,其命名规则是加前缀 subdir,所以我们看到有 subdir0。而数据块文件的命名规则,则是加前缀 blk_。还有一种文件,其前缀为 dncp_,是所谓的“拷贝文件”,dncp 应该是“datanode copy”的意思。然后,同样是用于数据块文件存储的目录,除我们已经看到的 rbw 和 finalized 之外,还会有 detach、lazypersist 和 tmp。

总之,DataStorage 代表着宿主系统上可以用来存储 HDFS 数据块的一个目录分支,HDFS 就以此作为一个文件卷,用一个 FsVolumeImpl 类对象为代表:

```
class FsVolumeImpl implements FsVolumeSpi {
    //The underlying volume used to store replica
    ] FsDatasetImpl dataset //所支撑的 FsDatasetImpl 对象
    ] String storageID // storageID 就是文件卷所在 Storage 的 ID
    ] StorageType storageType //可以是 DISK、SSD、ARCHIVE 或 RAM_DISK
    ] Map<String, BlockPoolSlice> bpSlices //本文件卷上所存储的所有 BlockPoolSlice
    ] File currentDir; // current 目录的路径
    ] DF usage //用 unix 的 df 命令获取关于文件卷的统计信息
    ] ThreadPoolExecutor cacheExecutor //用于本文件卷的 CachingTask 线程池
    // Per-volume worker pool that processes new blocks to cache
    ] FsVolumeImpl(FsDatasetImpl dataset, String storageID,
        File currentDir, Configuration conf, StorageType storageType) //构造方法
    > this.dataset = dataset
    > this.storageID = storageID
    > this.currentDir = currentDir // current 目录的路径
    > File parent = currentDir.getParentFile() //文件卷挂载点的路径
    > this.usage = new DF(parent, conf) //从执行 df 命令的结果中抽取信息
    > this.storageType = storageType // DISK、SSD、ARCHIVE 或 RAM_DISK
    > this.configuredCapacity = -1 //表示容量应根据实际情况通过计算产生
    > cacheExecutor = initializeCacheExecutor(parent) //创建 CachingTask 线程池
```

```

>>> ThreadFactory workerFactory = new ThreadFactoryBuilder()...build()
>>> ThreadPoolExecutor executor = new ThreadPoolExecutor(1, maxNumThreads, 60,
    TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>(), workerFactory)
>>> executor.allowCoreThreadTimeOut(true)
>>> return executor
] getCapacity()
> if (configuredCapacity < 0) { //如果 configuredCapacity<0,就根据实际情况加以计算
>+ remaining = usage.getCapacity() - reserved
>+ return remaining > 0?remaining : 0
> }
> return configuredCapacity //如果 configuredCapacity>=0,就采用设定的容量
] addBlockPool(String bpid, Configuration conf) //在文件卷中添加一个 BlockPoolSlice
> File bpdir = new File(currentDir, bpid) //在 current 目录下添加一个 BP 子目录
> BlockPoolSlice bp = new BlockPoolSlice(bpid, this, bpdir, conf) //创建一个对象
> bpSlices.put(bpid, bp) //将此 BlockPoolSlice 添加到 bpSlices 集合中

```

FsVolumeImpl 代表着 HDFS 落实在宿主系统中的一个文件卷,这个文件卷的载体可以是磁盘,也可以是 SSD 卡,也可以是某种外储设备例如 U 盘,还可以是建立在内存中的 RamDisk,所以文件卷的 StorageType 可以是 DISK、SSD、ARCHIVE 或 RAM_DISK。

每个 FsVolumeImpl 有个线程池 cacheExecutor,这些线程都是用来运行 CachingTask 的,目的在于从文件卷中装载数据块文件加以缓存,以提供内存中的缓冲存储,提高读出效率,对此后面还要结合具体情景加以介绍。

如前所述,一个文件卷中可以存储着属于多个 BlockPoolSlice 的数据块,bpSlices 就是这些 BlockPoolSlice 的集合。但是这并不意味着一个 BlockPoolSlice 的所有数据块都必须存储在同一个文件卷中。

那么 FsVolumeImpl 与 DataStorage 是什么关系呢? FsVolumeImpl 是从 HDFS 的视角来代表和管理一个文件卷,而 DataStorage 则是从设备和介质的视角来代表和管理一个(虚拟的)存储设备,一个设备上可以有多个文件卷。

当然,一个 DataNode 上可以有多个文件卷,所以还得有个全局性的对象来代表和管理本节点上所有数据块的集合。这就是 FsDatasetImpl。

```

class FsDatasetImpl implements FsDatasetSpi<FsVolumeImpl>{{
] DataNode datanode //在哪一个 DataNode 上
] DataStorage dataStorage //一个 DataStorage,代表着节点上所有的数据块
] FsVolumeList volumes // FsVolumeImpl 的 List,节点上可以有多个文件卷
] Map<String, DatanodeStorage> storageMap //从 Storage ID 到 DatanodeStorage 的 MAP,
    // DatanodeStorage 反映一个 DataNode 上 Storage 的状态信息
] ReplicaMap volumeMap //用于数据块复份的管理
    //根据 Blockpool ID 可以找到从 blockId 到具体 ReplicaInfo 的 MAP
] FsDatasetAsyncDiskService asyncDiskService
    //这是一个线程池,每个线程负责一个文件卷的异步磁盘操作

```

```

]] class ReplicaFileDeleteTask implements Runnable {}
]] ThreadGroup threadGroup //用来运行 SyncFileRange 和 ReplicaFileDeleteTask 的线程池
]] addVolume(File volume) //增加一个文件卷,从而增加一个线程
]] submitSyncFileRangeRequest(FsVolumeImpl volume, final FileDescriptor fd,
                               final long offset, final long nbytes, final int flags)
    > execute(volume.getCurrentDir(), new Runnable())
    >> run(0)

]] FsDatasetCache cacheManager //管理缓冲在内存中的数据块复份
]] cacheBlock(long blockId, String bpid, String blockFileName, ...) //为一数据块建立缓存
]] uncacheBlock(String bpid, long blockId) //去除一个数据块的缓存
]] class CachingTask implements Runnable {} //线程,负责建立数据块缓存
]] class UncachingTask implements Runnable {} //线程,负责去除数据块缓存
]] RamDiskReplicaTracker ramDiskReplicaTracker
    //跟踪存储在 RamDisk 中的数据块复份
]] Daemon lazyWriter // LazyWriter 线程,将 RamDisk 的内容存盘成为一个 checkpoint
]] RamDiskAsyncLazyPersistService asyncLazyPersistService
    //提供将 RamDisk 内容持久化存储的服务
]] ThreadGroup threadGroup //用来运行 ReplicaLazyPersistTask 的线程池
]] addVolume(File volume)
]] submitLazyPersistTask(String bpId, long blockId, long genStamp, long creationTime,
                          File metaFile, File blockFile, FsVolumeImpl targetVolume)
]] class ReplicaLazyPersistTask implements Runnable {}
]] getVolume(final ExtendedBlock b)
]] getStoredBlock(String bpid, long blkid)
]] fetchReplicaInfo()
]] getMetaDataInputStream(ExtendedBlock b)
]] addVolume(Collection<StorageLocation> dataLocations, Storage.StorageDirectory sd)
    > File dir = sd.getCurrentDir()
    > StorageType storageType = getStorageTypeFromLocations(dataLocations, sd.getRoot())
    > fsVolume = new FsVolumeImpl(this, sd.getStorageUuid(), dir, this.conf, storageType)
    > ReplicaMap tempVolumeMap = new ReplicaMap(this)
    > fsVolume.getVolumeMap(tempVolumeMap, ramDiskReplicaTracker)
    > volumeMap.addAll(tempVolumeMap)
    > volumes.addVolume(fsVolume)
    > s = new DatanodeStorage(sd.getStorageUuid(),
                            DatanodeStorage.State.NORMAL, storageType)
    > storageMap.put(sd.getStorageUuid(), s)
]] addVolumeAndBlockPool(Collection<StorageLocation> dataLocations,
                          Storage.StorageDirectory sd, final Collection<String> bpbids)
    > File dir = sd.getCurrentDir()

```

```

> StorageType storageType = getStorageTypeFromLocations(dataLocations, sd.getRoot())
> fsVolume = new FsVolumeImpl(this, sd.getStorageUuid(), dir, this.conf, storageType)
> tempVolumeMap = new ReplicaMap(fsVolume)
> for (final String bpid : bpids) {
>+ fsVolume.addBlockPool(bpid, this.conf)
>+ fsVolume.getVolumeMap(bpid, tempVolumeMap, ramDiskReplicaTracker)
> }
> volumeMap.addAll(tempVolumeMap)
> s = new DatanodeStorage(sd.getStorageUuid(),
                        DatanodeStorage.State.NORMAL, storageType)
> storageMap.put(sd.getStorageUuid(), s)
> asyncDiskService.addVolume(sd.getCurrentDir())
> volumes.addVolume(fsVolume)
] ReplicaInPipeline append(ExtendedBlock b, long newGS, long expectedBlockLen)
    //GS为 Generation Stamp 的缩写,相当于世代标记
] addVolumes(List<StorageLocation> volumes, final Collection<String> bpids)
    > dataLocations = DataNode.getStorageLocations(this.conf)
    > allStorageDirs = new HashMap<String, Storage.StorageDirectory>()
    > for (int idx = 0; idx < dataStorage.getNumStorageDirs(); idx++) {
    >+ Storage.StorageDirectory sd = dataStorage.getStorageDir(idx)
    >+ allStorageDirs.put(sd.getRoot().getCanonicalPath(), sd)
    > }
    > List<Thread> volumeAddingThreads = Lists.newArrayList()
    > for (int i = 0; i < volumes.size(); i++) {
    >+ Thread t = new Thread() {}
        ] run()
            > StorageLocation vol = volumes.get(idx)
            > String key = vol.getFile().getCanonicalPath()
            > addVolumeAndBlockPool(dataLocations, allStorageDirs.get(key), bpids)
            > successFlags[idx] = true
    >+ volumeAddingThreads.add(t)
    >+ t.start() //t.run()
    > }
    > for (Thread t : volumeAddingThreads) t.join()
    > setupAsyncLazyPersistThreads()
    > for (int i = 0; i < volumes.size(); i++) {
    >+ if (successFlags[i]) succeedVolumes.add(volumes.get(i))
    > }

```

FsDatasetImpl 代表着存储在一个 DataNode 上的所有数据块(复份)及其物理存储的集合,概念上与 DataStorage 比较接近,事实上 DataStorage 是 FsDatasetImpl 内部的一个成分。

但是除 `DataStorage` 之外 `FsDatasetImpl` 还提供了许多别的成分。首先,这些数据块可以存储在多个文件卷上,而 `FsDatasetImpl` 内部的 `FsVolumeList` 就是这些文件卷的集合,这样就把各个文件卷的 `FsVolumeImpl` 给整合了进来。然后是 `storageMap`,通过这个 MAP 凭 `StorageUuid` 就可以快速找到相应的 `DatanodeStorage`,这是 `DataNode` 向 `NameNode` 提交状态报告用的。还有 `volumeMap`,这是个 `ReplicaMap`,这是一个 MAP 的 MAP,在此 MAP 中用 `Blockpool ID` 可以找到从 `blockId` 到具体 `ReplicaInfo` 的 MAP。也就是说,先用 `bpId` 在 `volumeMap` 中找到一个 MAP,然后在这个 MAP 中用 `blockId` 找到这个块的复份。

`FsDatasetImpl` 还直接或间接地创建了好多线程,这些线程主要用于以下三个目的。

第一是磁盘的异步读写。磁盘文件的读写是比较慢的,如果是同步读写,即每次读写磁盘时都要阻塞等待,那效率就低了。比较好的办法是异步读写,把它交给专管磁盘读写的线程去完成。这里的 `asyncDiskService` 就是为此目的而设的线程池。

第二是缓存的管理。为提高 HDFS 文件读操作的效率,内存中要根据访问频度缓存一些数据块的映像,但是这个数据块映像的集合是动态变动的,需要根据 LRU 算法加以调度,这里的 `FsDatasetCache` 类对象 `cacheManager` 就是这么一个集合。

第三是 `RamDisk` 中数据块映像的持久化存储。如前所述,`NameNode` 在指引用户把数据写入 HDFS 时,会让用户把数据写入多个(通常是三个)`DataNode`,成为数据块的多个复份。但是一般这多个复份并非都要立即写入磁盘,因为磁盘的操作太慢,而是有个存储策略,比方说把其中的一份交给一个 `DataNode` 让它直接写入磁盘,另两份则分别交给两个 `DataNode`,让它们写入 `RamDisk`。这样,如果有别的用户马上就想要读取这些数据,则有两个 `DataNode` 立即就可从它们的 `RamDisk` 中读出,这时候需要把复份写入磁盘的那个 `DataNode` 可能还远未完成写入。然而写入 `RamDisk` 中的复份却不能永远留在 `RamDisk` 中,一来 `RamDisk` 的容量有限,二来还要考虑所在的 `DataNode` 可能会损坏或需要关机或重启。所以写入 `RamDisk` 的数据块映像最终还是得要转移到磁盘上加以永久存储,这就是数据块复份持久化的问题。另外,与 `NameNode` 上的目录系统一样,数据块映像也有建立可供恢复的版本即 `checkpoint` 的问题。为磁盘上的数据块文件建立 `checkpoint` 很简单,只要另建一个目录并拷贝一个副本到那个目录中就行了;但是为 `RamDisk` 中的数据文件建立 `checkpoint` 就要麻烦一些,因为 `checkpoint` 的目录不能建在 `RamDisk` 中,而要建在磁盘上。这里的 `RamDiskAsyncLazyPersistService` 类对象 `asyncLazyPersistService`,以及 `lazyWriter` 线程,就都是为有关 `RamDisk` 的这些目的服务的。

最后,`FsDatasetImpl` 还提供了一些它那个层次上的操作方法,相当于一个小小函数库。

所以尽管 `FsDatasetImpl` 与 `DataStorage` 都在某种程度上代表着一个 `DataNode` 上的所有数据块(复份)及其物理存储,二者还是有挺大差别的,但是 `DataStorage` 无疑是基础。

大体上知道了这些,我们可以看一下具体子目录中的数据块文件是什么样的。这是在 `hdfs/dn/current/BP-2085968454-127.0.1.1-1426747700215/current/finalized/subdir0/subdir0`,执行 `ls` 命令的结果:

```
-rw-r--r-- 1 root root      356 2015-03-19 14:53 blk_1073741849
-rw-r--r-- 1 root root      11 2015-03-19 14:53 blk_1073741849_1025.meta
-rw-r--r-- 1 root root 130565 2015-03-19 14:53 blk_1073741850
-rw-r--r-- 1 root root 1031 2015-03-19 14:53 blk_1073741850_1026.meta
```



```
-rw-r--r-- 1 root root 104491 2015-03-19 14:53 blk_1073741851
-rw-r--r-- 1 root root 827 2015-03-19 14:53 blk_1073741851_1027.meta
```

这里有三个块文件,每个在宿主机上实际存储着两个文件,例如 blk_1073741849 与 blk_1073741849_1025.meta,前者是数据本身,后者是“元数据”文件,里面记载着数据文件的结构信息,文件名带有后缀.meta。元文件都是很小的,数据文件的大小(即数据块的大小)则可以通过 config 文件加以设置,默认为 64MB,我的机器上设置成 128KB。不过从 ls 命令的输出可见,这三个数据文件都还不到 128KB,但是却已经在 finalized 分支下面,这并不意味着此后就不能再往这三个文件中写了,而只是说文件已经关闭,已经写入的内容就封存不能变了,但是仍可在文件尾部接着写,直至达到预定的大小,那时候就得另起一块了。

我们在前面看到的从 StorageInfo、Storage、BlockPoolSliceStorage,到 DataStorage,乃至 FsVolumeImpl 和 FsDatasetImpl,虽然也涉及了块文件,但主要都是从存储架构的角度着眼的,我们还没有看到程序中代表着数据块的数据结构,即“类”。当然,我们需要有这么一个类,这就是 Block:

```
class Block implements Writable, Comparable<Block> {}
] static final String BLOCK_FILE_PREFIX = "blk_"
] static final String METADATA_EXTENSION = ".meta"
] long blockId //块号
] long numBytes //块的当前大小
] long generationStamp //世代标记,相当于版本号
] getBlockName()
> return BLOCK_FILE_PREFIX + String.valueOf(blockId)
] ... //还有一些别的操作方法,包括构造方法
```

这个类定义了两个常数分别用于数据块文件名的前缀和元数据(meta)文件的后缀,但是类的结构成分中没有文件名,而只有块号 blockId,因为文件名是可以按固定的规则根据块号生成的。这样,从数据(不包括操作)的角度看,块号、当前长度和世代标记(相当于“重大版本号”)这三个要素的组合就是对于 Block 的抽象,代表着一个 Block。

不过,DataNode 所以为的“块”,从 NameNode 看来只是这个块的一个复份,即 Replica。可想而知,Replica 应该是对 Block 类的扩充,因为 Replica 比 Block 更具体:

```
abstract class ReplicaInfo extends Block implements Replica {}
//This class is used by datanodes to maintain meta data of its replicas.
] FsVolumeSpi volume //volume where the replica belongs,所属的文件卷
] File baseDir //directory where block & meta files belong,所在的宿主目录
] Map<String, File> internedBaseDirs //以供快速找到所在的宿主目录
] getBlockFile() //Get the full path of this replica's data file
> return new File(getDir(), getBlockName())
] getMetaFile() //Get the full path of this replica's meta file
> return new File(getDir(),
DatanodeUtil.getMetaName(getBlockName(), getGenerationStamp()))
```

```

    ] parseBaseDir(File dir)
    ] class ReplicaDirInfo{}
```

与 Block 相比,这里增加了 volume 和 baseDir 两个要素。这就是说,一个逻辑上的 Block,存储在某个具体文件卷的某个目录下面,就成为一个具体的 Replica。当然,这里忽略了一个要素,就是在哪一个节点上;但是从 DataNode 的角度看这不是问题,因为这当然是在本节点上。

注意这只是个抽象类,具体的 Replica 还有更多的约束条件,还要加以扩充:

```

class FinalizedReplica extends ReplicaInfo {}
class ReplicaUnderRecovery extends ReplicaInfo {}
class ReplicaWaitingToBeRecovered {}
class ReplicaInPipeline extends ReplicaInfo {}
```

就是说,为处于不同状态的 Replica 定义了不同的类。那为什么不是统一定义一个 Replica 类,而在里面加上一个表示状态的字段呢?这应该是因为:处于不同状态的 Replica,它们的操作方法实在差别太大了,以至于把它们统在一起是没有多大意义的了。这是后话,现在暂且搁置不论。

回到前面所看到的那三个块的数据文件和 meta 文件。DataNode 只知道这是三个块,至于这三个块分别属于哪一个或哪几个 HDFS 文件,DataNode 是不关心的。所以 DataNode 就好比是个元器件仓库,它不关心你从仓库提走的元器件是用在哪一部机器上。

于是我们就有了两个问题。第一,这几个块文件当初是怎么建立的,里面的数据是怎么写进去的?或者,在现有的这个基础上,如果又有一个新的数据块要存储到这个 DataNode 上,将是怎么一个过程?第二,假定现有的这几个块已经存储在磁盘上,机器开机时 DataNode 怎样根据磁盘上的内容建立起内存中的那些数据结构(对象),来管理这些磁盘文件?

我们先考察第一个问题,即块文件的创建和写入。这里我们暂且不管 DataNode 与 NameNode 之间,以及 DataNode 与 DataNode 之间的通信和传输,而只是从 DataNode 接收到一个数据块的时候开始。

12.3 RamDisk 复份的持久化存储

DataNode 把接收到的数据块复份写入本地的宿主文件系统并 finalize,该复份就正式存储在本节点的某个文件卷上了。但是这里又有个特例,就是:存储类型为 RAM_DISK 的数据块复份,所写入的文件卷其实是建立在内存中,不是永久性的,什么时候节点一断电,内容就“挥发”了。再说,RAM_DISK 的容量也很有限,很快就会被用完。我们之所以把复份写入 RAM_DISK,主要是为了快速完成写入,另一方面对快速读出当然也有好处。但是只留在 RAM_DISK 中终非长久之计,总得将其转入持久性的介质才好。不过一定要马上就做持久化存储也不好,因为那样可能会使 DataNode 的负担在一段时间内变得很重,而在别的时间里却又可能很空。比较好的办法是给予一定的弹性,使 DataNode 可以推迟这样的持久化存储。

为此,HDFS 提供了一种异步的、可以推迟进行的持久化存储机制和服务,那就是 RamDiskAsyncLazyPersistService。此种服务的使用者当然就是 DataNode 本身,更具体地说

是 FsDatasetImpl。

FsDatasetImpl 内部定义了一个 LazyWriter 类, 然后在其构造函数内创建了一个 LazyWriter 对象, 实际上是一个线程, 这个线程就是此种事务的“专管员”, 是从 FsDatasetImpl 被创建之后就有的。所以我们从 LazyWriter 的由来开始:

```
class FsDatasetImpl implements FsDatasetSpi<FsVolumeImpl> {}
] Daemon lazyWriter
] FsDatasetImpl(DataNode datanode, DataStorage storage, Configuration conf)
  > ...
  > lazyWriter = new Daemon(new LazyWriter(conf))
  > lazyWriter.start()
```

可见, DataNode 在创建 FsDatasetImpl 的时候就间接创建了 LazyWriter 线程。

```
class FsDatasetImpl.LazyWriter implements Runnable {}
] LazyWriter(Configuration conf)    //构造方法
  > this.checkpointerInterval = conf.getInt(...) //从配置文件获取操作间隔长度, 默认为 60 秒
  > this.lowWatermarkFreeSpacePercentage = conf.getFloat(...)
                                     //从配置文件获取要求保持的空闲存储空间占比, 默认为 10%
  > this.lowWatermarkFreeSpaceBytes = conf.getLong(...)
                                     //从配置文件获取要求保持空闲的存储空间, 默认为一个块的大小(64MB 或 128MB)
] run()
  > while (fsRunning && shouldRun) { //线程的主循环
  >+ numSuccessiveFailures = saveNextReplica()?0 : (numSuccessiveFailures + 1)
  >+ evictBlocks()
  >+ if (numSuccessiveFailures >= ramDiskReplicaTracker.numReplicasNotPersisted()) {
  >++ Thread.sleep(checkpointerInterval * 1000) //暂时无事可干就睡一会儿
  >++ numSuccessiveFailures = 0
  >+ }
  > }
] saveNextReplica()
] evictBlocks()
```

这个线程的主循环中就是两大步, saveNextReplica() 和 evictBlocks(), 前者从 RAMDISK 中把若干复份, 包括其块文件和元文件, 拷贝到磁盘上, 后者则从 RAMDISK 中删除该复份。

```
[FsDatasetImpl.LazyWriter.run() > saveNextReplica()]
```

```
FsDatasetImpl.LazyWriter.saveNextReplica()
> RamDiskReplica block = ramDiskReplicaTracker.dequeueNextReplicaToPersist()
                                     //从 RamDiskReplicaTracker 的队列中解下一个等待被持久存储的复份
>> while (replicasNotPersisted.size() != 0) { //循环直至 replicasNotPersisted.size() 为 0
>>+ RamDiskReplicaLru ramDiskReplicaLru = replicasNotPersisted.remove()
```

```

// replicasNotPersisted 队列中是按 LRU 算法裁定应退出 RAMDISK 的复份
>>>+ Map<Long, RamDiskReplicaLru> replicaMap =
        replicaMaps.get(ramDiskReplicaLru.getBlockPoolId())
        //根据其 bpid 找到该 BP 的 replicaMap
>>>+ if (replicaMap!= null && replicaMap.get(ramDiskReplicaLru.getBlockId())!= null) {
>>>+ return ramDiskReplicaLru //核查无误,这就是要转入二线的数据块复份
>>>+ } //找到一个就返回,要不然就继续找下一个
>> } //end while,
>> return null //没有就返回 null
> if (block!= null) { //如果有的话
>+ replicaInfo = volumeMap.get(block.getBlockPoolId(), block.getBlockId())
        //从 volumeMap 中获取关于这个复份的更详细的信息
>+ if (replicaInfo!= null && replicaInfo.getVolume().isTransientStorage()) {
        //确认这个复份是在过渡性存储介质(RAM_DISK)上
>+ targetVolume = volumes.getNextVolume(StorageType.DEFAULT,
        replicaInfo.getNumBytes()) //找到一个可以容纳这个复份的文件卷
>+ ramDiskReplicaTracker.recordStartLazyPersist(block.getBlockPoolId(),
        block.getBlockId(), targetVolume)
        == RamDiskReplicaTracker.recordStartLazyPersist() //开始着手该复份的久储操作
>+> Map<Long, RamDiskReplicaLru> map = replicaMaps.get(bpid)
>+> RamDiskReplicaLru ramDiskReplicaLru = map.get(blockId)
>+> ramDiskReplicaLru.setLazyPersistVolume(checkpointVolume) //设置目标文件卷
>+> asyncLazyPersistService.submitLazyPersistTask(block.getBlockPoolId(),
        block.getBlockId(), replicaInfo.getGenerationStamp(), block.getCreationTime(),
        replicaInfo.getMetaFile(), replicaInfo.getBlockFile(), targetVolume)
        == RamDiskAsyncLazyPersistService.submitLazyPersistTask(...)
>+> File lazyPersistDir = targetVolume.getLazyPersistDir(bpId) //按 bpid 找到应去的目录
>+> if (!lazyPersistDir.exists() && !lazyPersistDir.mkdirs()) { //若不存在就创建
>+>+ IOException("LazyWriter fail to find or create lazy persist dir: " ...) //创建失败就发异常
>+>+ }
>+> lazyPersistTask = new ReplicaLazyPersistTask(bpId, blockId, genStamp,
        creationTime, blockFile, metaFile, targetVolume, lazyPersistDir) //创建一个 runnable
>+> execute(targetVolume.getCurrentDir(), lazyPersistTask) //安排一个线程加以执行
        == ReplicaLazyPersistTask.run()
>+>> targetFiles[] = FsDatasetImpl.copyBlockFiles(blockId, genStamp,
        metaFile, blockFile, lazyPersistDir)
        //拷贝这个复份的数据文件和元数据文件
>+>> datanode.getFSDataset().onCompleteLazyPersist(bpId, blockId, ...)
        //与 recordStartLazyPersist()相对应,用于善后和统计
>+ }

```

```

> } //end if (block!= null)
> succeeded = true    //能到达这里而不发生异常,即是成功
> return succeeded

```

可见,saveNextReplica()是从 RamDiskReplicaTracker 的 replicasNotPersisted 队列中获取需要被转移到磁盘上的数据块复份;从 RAMDISK 转移一个数据块复份到磁盘上,就是从 RAMDISK 这个文件卷上拷贝其块文件和元文件到磁盘上的文件卷中。至于 RAMDISK 上的原件,则随时可以删除。

注意,saveNextReplica()是个异步的过程,实际的文件拷贝是另外创建线程加以完成的,所以 saveNextReplica()不会被阻塞,立即就能返回。从 saveNextReplica()返回后,下一步就是 evictBlocks(),目的是从 RAMDISK 上删除一些数据块复份。

RamDiskReplicaTracker 中有两个队列,一个是前面看到的 replicasNotPersisted, RAMDISK 上所有尚未持久存储的复份都按 LRU 算法决定的先后排在那个队列中,最近最少用到的复份排在最前面,需要将一些复份转入持久存储时就从该队列的头上取。将一个复份拷贝到磁盘上后就把这个复份转到另一个队列 replicasPersisted 中,这是已经持久存储,可以从 RAMDISK 上删除的复份。

```
[FsDatasetImpl.LazyWriter.run() > evictBlocks()]
```

```
FsDatasetImpl.LazyWriter.evictBlocks()
```

```

> while (iterations++ < MAX_BLOCK_EVICTIONS_PER_ITERATION
           && transientFreeSpaceBelowThreshold()) {
           //每次只做有限次循环,并且一旦 RAMDISK 的“水位”够高了就停止
>+ RamDiskReplica replicaState = ramDiskReplicaTracker.getNextCandidateForEviction()
           //从 RamDiskReplicaTracker 的队列中解下一个可以被删除的复份
>+> Iterator<RamDiskReplicaLru> it = replicasPersisted.values().iterator()
>+> while (it.hasNext()) {
>+>+ RamDiskReplicaLru ramDiskReplicaLru = it.next()
>+>+ it.remove()
>+>+ Map<Long, RamDiskReplicaLru> replicaMap =
           replicaMaps.get(ramDiskReplicaLru.getBlockPoolId())
>+>+ if (replicaMap!= null && replicaMap.get(ramDiskReplicaLru.getBlockId())!= null) {
>+>++ return ramDiskReplicaLru
>+>+ }
>+> }
>+> return null
>+ if (replicaState == null) break    //没有了,结束循环
>+ String bpid = replicaState.getBlockPoolId()
>+ replicaInfo = getReplicaInfo(replicaState.getBlockPoolId(), replicaState.getBlockId());
           Preconditions.checkState(replicaInfo.getVolume().isTransientStorage());
>+ File blockFile = replicaInfo.getBlockFile()    //这是 RAMDISK 上的块文件

```

```

>+ File metaFile = replicaInfo.getMetaFile()           //这是 RAMDISK 上的元文件
>+ ramDiskReplicaTracker.discardReplica( /* 从 replicasPersisted 队列中删去
      replicaState.getBlockPoolId(), replicaState.getBlockId(), false)
>+ // Move the replica from lazyPersist/ to finalized/ on target volume
>+ bpSlice = replicaState.getLazyPersistVolume().getBlockPoolSlice(bpid)
>+ File newBlockFile = bpSlice.activateSavedReplica(replicaInfo,
      replicaState.getSavedMetaFile(), replicaState.getSavedBlockFile())
>+ newReplicaInfo = new FinalizedReplica(replicaInfo.getBlockId(),
      replicaInfo.getBytesOnDisk(), replicaInfo.getGenerationStamp(),
      replicaState.getLazyPersistVolume(), newBlockFile.getParentFile())
>+ volumeMap.add(bpid, newReplicaInfo) // Update the volumeMap entry.
      //用它替换 volumeMap 中老的 ReplicaInfo,以后这个复份就在磁盘上了
>+ ExtendedBlock extendedBlock = new ExtendedBlock(bpid, newReplicaInfo)
>+ datanode.getShortCircuitRegistry().processBlockInvalidation(
      ExtendedBlockId.fromExtendedBlock(extendedBlock))
>+ datanode.notifyNamenodeReceivedBlock(extendedBlock, null,
      newReplicaInfo.getStorageUuid()) //通知 NameNode
>+ if (blockFile.delete()||!blockFile.exists()) { //删除 RAMDISK 上的块文件
>++ ((FsVolumeImpl) replicaInfo.getVolume()).decDfsUsed(bpid, blockFileUsed)
>++ if (metaFile.delete()||!metaFile.exists()) { //删除 RAMDISK 上的元文件
>+++ ((FsVolumeImpl) replicaInfo.getVolume()).decDfsUsed(bpid, metaFileUsed)
>+++ }
>++ // If deletion failed then the directory scanner will cleanup the blocks eventually.
>+ }
> }

```

这样,saveNextReplica()和 evictBlocks()合在一起,就把 RAMDISK 上应该转入持久存储的数据块复份转移到磁盘上去了。

可是怎么知道 RAMDISK 上的哪些复份应该转移到磁盘上呢?其实我们在上面已经看到,那些复份的 RamDiskReplica 对象都取自 ramDiskReplicaTracker 的 replicasNotPersisted 队列中。显然这与 ramDiskReplicaTracker 有关。FsDatasetImpl 中的 ramDiskReplicaTracker 是个 RamDiskReplicaTracker 类的对象,然而 RamDiskReplicaTracker 是个抽象类,实际创建的必定是经过扩充的某类对象。我们先看一下这个抽象类的摘要:

```

abstract class RamDiskReplicaTracker{
}
class RamDiskReplica implements Comparable<RamDiskReplica>{
}
String bpid
long blockId
File savedBlockFile
File savedMetaFile
long creationTime

```



```

]] boolean isPersisted
]] FsVolumeSpi ramDiskVolume //RAM_DISK volume that holds the original replica.
]] FsVolumeImpl lazyPersistVolume //Persistent volume that holds or will hold the saved replica.
] getInstance(final Configuration conf, final FsDatasetImpl fsDataset)
    //受 FsDatasetImpl 调用以创建实体的对象
    > trackerClass = conf.getClass(..., RamDiskReplicaLruTracker.class,
                                   RamDiskReplicaTracker.class)
    //从配置文件获取扩充了 RamDiskReplicaTracker 的
    //类的名称,默认为 RamDiskReplicaLruTracker
    > tracker = ReflectionUtils.newInstance(trackerClass, conf) //创建一个新的对象
    > tracker.initialize(fsDataset) //初始化
    >> this.fsDataset = fsDataset

```

在 FsDatasetImpl 的构造方法中是通过 RamDiskReplicaTracker.getInstance() 创建 ramDiskReplicaTracker 的,所以具体创建什么类的对象取决于配置,未加设置则默认为 RamDiskReplicaLruTracker,实际上也就是 RamDiskReplicaLruTracker:

```

class RamDiskReplicaLruTracker extends RamDiskReplicaTracker {}
] class RamDiskReplicaLru extends RamDiskReplica {}
]] long lastUsedTime //比 RamDiskReplica 多了个最后使用时间,这是 LRU 算法的依据
] Map<String, Map<Long, RamDiskReplicaLru>> replicaMaps
] Queue<RamDiskReplicaLru> replicasNotPersisted
    //Queue of replicas that need to be written to disk.
] TreeMultimap<Long, RamDiskReplicaLru> replicasPersisted
    //Map of persisted replicas ordered by their last use times.
- - - - - 以上为数据结构部分 - - - - -
] addReplica(final String bpid, final long blockId, final FsVolumeImpl transientVolume)
] touch(final String bpid, final long blockId)
] recordStartLazyPersist(final String bpid, final long blockId, FsVolumeImpl checkpointVolume)
] recordEndLazyPersist(final String bpid, final long blockId, final File[] savedFiles)
] dequeueNextReplicaToPersist()
] getNextCandidateForEviction()
] discardReplica(String bpid, long blockId, boolean deleteSavedCopies)

```

可想而知,RamDiskReplicaLruTracker 根据 LRU 算法调整 RAMDISK 上那些数据块复份在 replicasNotPersisted 队列中的位置,使最近最少使用的复份排在队列的前面。至于具体的算法和实现这里就不深入下去了,有需要或兴趣的读者可以自行阅读。

此外,前面在 saveNextReplica()中还涉及了 RamDiskAsyncLazyPersistService 类的对象 asyncLazyPersistService,特别是用到了它的线程池,及其内部定义的 ReplicaLazyPersistTask 类。下面是 RamDiskAsyncLazyPersistService 类的摘要:

```

class RamDiskAsyncLazyPersistService {}
] addExecutorForVolume(final File volume)

```

```

> threadFactory = new ThreadFactory()
> executor = new ThreadPoolExecutor(...,
                                new LinkedBlockingQueue<Runnable>(), threadFactory)
> executors.put(volume, executor)
] addVolume(File volume)
    > ThreadPoolExecutor executor = executors.get(volume)
    > addExecutorForVolume(volume)
] submitLazyPersistTask(String bpId, long blockId, ..., FsVolumeImpl targetVolume)
    > File lazyPersistDir = targetVolume.getLazyPersistDir(bpId)
    > if (!lazyPersistDir.exists() &&!lazyPersistDir.mkdirs()) {
    >+ IOException("LazyWriter fail to find or create lazy persist dir: " ...)
    > }
    > lazyPersistTask = new ReplicaLazyPersistTask(bpId, blockId, genStamp,
                                creationTime, blockFile, metaFile, targetVolume, lazyPersistDir)
                                //新创一个线程,专门用来完成一个复份的转储
    > execute(targetVolume.getCurrentDir(), lazyPersistTask) //启动这个新创的线程
] class ReplicaLazyPersistTask implements Runnable {}

```

作为 FsDatasetImpl 的内部成分, asyncLazyPersistService 和 asyncDiskService 都是在创建 FsDatasetImpl 对象时的构造方法中创建的。显然,对于 DataNode, FsDatasetImpl 是个很重要的成分,因为这个成分代表着存储在这个节点上的数据集合。

与 NameNode 相比,就 HDFS 文件系统的结构而言 DataNode 处于被动和从属的地位;但是就二者之间的互动而言,DataNode 却总是处于主动的地位;而且 DataNode 中的成分和部件不少,所以看其 startDataNode() 令人有眼花缭乱之感。下面是作者整理的一个粗线条的流程摘要,从中既可看出这个过程的大致流程和层次,又可看出一些部件之间是什么关系,对我们理解 DataNode 的结构应该有些帮助:

```

startDataNode()
> blockPoolManager.refreshNamenodes()
>> doRefreshNamenodes()
>>> BlockPoolManager.createBPOS()
>>>> BPOfferService{} //对于集群中的每一个 NameNode 都要有个 BPOS 对象
>>>>> BPServiceActor.run() //其中的 BPServiceActor 是个线程
>>>>>> BPServiceActor.offerService()
>>>>>>> DataBlockScanner.addBlockPool(String blockPoolId)
//为每个 NN 都提供一个 BlockPool,但是在具体 DN 上是 BlockPoolSlice
>>>>>>>> new BlockPoolSliceScanner(blockPoolId, datanode, dataset, conf)
//每个 BlockPoolSlice 都有个 BlockPoolSliceScanner
>>>>>>> connectToNNAndHandshake() //连接到这个 BPOS 负责联络的 NameNode
>>>>>>>> BPOfferService.verifyAndSetNameSpaceInfo()
>>>>>>>>> initBlockPool() //属于这个 NameNode 的 BlockPool,其实是 BlockPoolSlice

```

```

>>>>>>>>> initPeriodicScanners() //周期性的扫描器有两种
>>>>>>>>> DataNode.initDirectoryScanner()
>>>>>>>>> new DirectoryScanner(data, conf) //目录扫描器
>>>>>>>>> DataNode.initDataBlockScanner()
>>>>>>>>> new DataBlockScanner() //数据块扫描器
>>>>>>>>> DataNode.initStorage() //存储的设备
>>>>>>>>> FsDatasetFactory.newInstance(this, storage, conf)
>>>>>>>>> new FsDatasetImpl(datanode, storage, conf) //存储着的数据块说明的集合

```

这里 FsDatasetImpl 代表着一个数据块集合, 这些数据存储在一个虚拟的存储设备上, 以一个 DataStorage 类的对象为代表:

```

class DataStorage extends Storage {}
] static String BLOCK_SUBDIR_PREFIX = "subdir"
] static String BLOCK_FILE_PREFIX = "blk_"
] static String COPY_FILE_PREFIX = "dncp_"
] static String STORAGE_DIR_DETACHED = "detach"
] static String STORAGE_DIR_RBW = "rbw"
] static String STORAGE_DIR_FINALIZED = "finalized"
] static String STORAGE_DIR_LAZY_PERSIST = "lazypersist"
] static String STORAGE_DIR_TMP = "tmp"
] String datanodeUuid
] Map<String, BlockPoolSliceStorage> bpStorageMap
] writeAll(Collection<StorageDirectory> dirs)
    > this.layoutVersion = getServiceLayoutVersion()
    > for (StorageDirectory dir : dirs) writeProperties(dir)
] addStorageLocations (DataNode datanode, NamespaceInfo nsInfo,
    Collection<StorageLocation> dataDirs, StartupOption startOpt)
] recoverTransitionRead(DataNode datanode, NamespaceInfo nsInfo,
    Collection<StorageLocation> dataDirs, StartupOption startOpt)
] makeBlockPoolDataDir(Collection<File> dataDirs, Configuration conf)
] format(StorageDirectory sd, NamespaceInfo nsInfo, String datanodeUuid)
] doTransition (DataNode datanode, StorageDirectory sd,
    NamespaceInfo nsInfo, StartupOption startOpt)
//Analyze which and whether a transition of the fs state is required and perform it if necessary.
] doUpgrade(DataNode datanode, StorageDirectory sd, NamespaceInfo nsInfo)
    //Upgrade — Move current storage into a backup directory, and hardlink
    //all its blocks into the new current directory.
] doRollback( StorageDirectory sd, NamespaceInfo nsInfo)
    //Rolling back to a snapshot in previous directory by moving it to current directory.
] doFinalize(StorageDirectory sd) //Finalize procedure deletes an existing snapshot.

```

```

] finalizeUpgrade(String bpID)      //Finalize the upgrade for a block pool
] linkAllBlocks(DataNode datanode, File fromDir, File fromBbwDir, File toDir)
                                   //Hardlink all finalized and RBW blocks in fromDir to toDir
] addBlockPoolStorage(String bpID, BlockPoolSliceStorage bpStorage)
  > if (!this.bpStorageMap.containsKey(bpID)) {
  >+ this.bpStorageMap.put(bpID, bpStorage)
  > }

```

之所以说是虚拟的存储设备,是因为数据块实际上是存储在宿主文件系统的某些目录中,并且在代表着同一个 BlockPool 的目录下也还有诸如 rbw、finalized、detach、tmp 等子目录,这些子目录各有用处。比方说,一个文件中的最后一块,一般都不是一次性写入的,而可能会分多次写入,所以这时候就把它放在 rbw 子目录下,rbw 是“replica being written”的意思。到这一块写满了,并且后面已经另起一块,这就不会再有变动了,此时就可以把它移到 finalized 子目录中。

12.4 目录扫描线程 DirectoryScanner

前面讲过,在 HDFS 中并不是由 NameNode 根据持久存储的文件系统元数据确定一个块的几个备份分别存储在哪一些节点上,再让访问文件的客户前去读取;NameNode 根本就不持久存储数据块的位置信息,而是由 DataNode 通过心跳提交报告,说明当地有着一些什么数据块(bpid 和 blockId)的备份。可是,DataNode 怎么知道自己当地有些什么呢?显然,唯一的办法就是自己清点。所以,DataNode 上有个 DirectoryScanner 线程,过一会儿就来扫描清点一下。扫描的结果主要用于向 NameNode 提交报告,一方面也可用于本节点上存储系统的维护。

DirectoryScanner 是 DataNode 的两种周期性扫描器之一(另一种是 DataBlockScanner,前者的目的是发现本地有些什么数据块,后者的目的是验证这些数据块完好无损)。DataNode 在通过 initBlockPool()建立本地的块池时会在 initPeriodicScanners()中调用 initDirectoryScanner():

```
[DataNode.initBlockPool() > initPeriodicScanners() > initDirectoryScanner()]
```

```
DataNode.initDirectoryScanner()
```

```

> if (directoryScanner != null) return //已经创建,返回
> if (conf.getInt(DFS_DATANODE_DIRECTORYSCAN_INTERVAL_KEY,
    DFS_DATANODE_DIRECTORYSCAN_INTERVAL_DEFAULT) < 0) {
>+ reason = "verification is turned off by configuration"
> } else if ("SimulatedFSDataset".equals(data.getClass().getSimpleName())){
>+ reason = "verification is not supported by SimulatedFSDataset"
> }
> if (reason == null) { //无失败原因,就是成功
>+ directoryScanner = new DirectoryScanner(data, conf) //创建 DirectoryScanner 线程
    //参数 data,就是 DataNode.data,是个 FsDatasetImpl 对象

```

```
>+ directoryScanner.start() //启动该线程
> } else {
>+ LOG.info("Periodic Directory Tree Verification scan is disabled because " + reason)
> }
```

首先以“dfs.datanode.directoryscan.interval”为配置项名称从配置块中获取对扫描间隔的设置,默认为 21600 秒,即 6 小时。但是也可以把这个配置项设成-1,那就是不要扫描。

只要此项功能没有被关闭,DataNode 就会创建 DirectoryScanner 线程。我们看一下这个类的摘要:

```
class DirectoryScanner implements Runnable {}
] FsDatasetSpi<?> dataset //实际是个 FsDatasetImpl 对象
] ExecutorService reportCompileThreadPool //用于编制报告的线程池
] ScheduledExecutorService masterThread //主线程,实际上就是 DirectoryScanner
] ScanInfoPerBlockPool diffs = new ScanInfoPerBlockPool()
    //ScanInfoPerBlockPool 是对 HashMap<String, LinkedList<ScanInfo>>> 的扩充
    //用来盛放当地数据块存储的变化
] DirectoryScanner(FsDatasetSpi<?> dataset, Configuration conf) //构造方法
> this.dataset = dataset
> interval = conf.getInt( //从配置文件获取扫描间隔时间,默认 21600 秒 */
    DFSConfigKeys.DFS_DATANODE_DIRECTORYSCAN_INTERVAL_KEY, ...)
> scanPeriodMsecs = interval * 1000L; //换算成毫秒
> threads = conf.getInt( //从配置文件获取线程池大小,默认为 1 */
    DFSConfigKeys.DFS_DATANODE_DIRECTORYSCAN_THREADS_KEY, ...)
    //创建线程池
> reportCompileThreadPool = Executors.newFixedThreadPool(threads,
    new Daemon.DaemonFactory())
> masterThread = new ScheduledThreadPoolExecutor(1, new Daemon.DaemonFactory())
    //创建主线程,因为 DirectoryScanner 是个 Runnable,需要有个线程来加以执行
] start()
> masterThread.scheduleAtFixedRate(this, offset, scanPeriodMsecs,
    TimeUnit.MILLISECONDS) //设置主线程运行间隔
] run() //Runs "reconcile()" periodically under the masterThread.
> reconcile()
>> scan() //先扫描,所得差异在 diffs 集合中
>> for (Entry<String, LinkedList<ScanInfo>>> entry : diffs.entrySet()) {
    //对于 diffs 集合中的每一个 diff 链
>>>+ String bpid = entry.getKey() //diffs 集合中的每一项都是针对一个 BlockPool 的
>>>+ LinkedList<ScanInfo> diff = entry.getValue() //取其 ScanInfo 链表部分
>>>+ for (ScanInfo info : diff) { //对于这个 diff 链中的每一个 ScanInfo
>>>++ dataset.checkAndUpdate(bpid, info.getBlockId(),
```

```

        info.getBlockFile(), info.getMetaFile(), info.getVolume())
    == FsDatasetImpl.checkAndUpdate(...)
>>+ } //end for (ScanInfo info : diff)
>> } // end for (Entry<String, LinkedList<ScanInfo>>> entry : diffs.entrySet())
>> if (!retainDiffs) clear()
] scan()    //见后
] getDiskReport()    //在 scan() 中被调用
] addDifference(LinkedList<ScanInfo> diffRecord, Stats statsRecord, ScanInfo info)
] addDifference(LinkedList<ScanInfo> diffRecord, Stats statsRecord,
                long blockId, FsVolumeSpi vol)
] class ReportCompiler implements Callable<ScanInfoPerBlockPool> {}
    //getDiskReport() 中创建的各个 reportCompileThreadPool 线程, 每个文件卷一个
] class ScanInfoPerBlockPool extends HashMap<String, LinkedList<ScanInfo>>> {}
] class ScanInfo implements Comparable<ScanInfo> {}
    //Tracks the files and other information related to a block on the disk
]] long blockId
]] String blockSuffix //The block file path, relative to the volume's base directory.
                        //could be the full path of the block file.
]] String metaSuffix //The suffix of the meta file path relative to the block file.
]] FsVolumeSpi volume
]] long blockFileLength

```

DirectoryScanner 是个 Runnable, 需要有个线程来加以执行, 即调用它的 run() 函数, 所以这里有个主线程, 每隔一定时间调用一次它的 run()。

每当受到调用时, 这个 run() 函数唯一的操作就是 reconcile(), 而 reconcile() 则先调用 scan() 扫描当地的文件系统, 将文件系统中见到的块文件集合与已知存放在本地的数据块集合相比较, 得出二者的差异, 存放在 diffs 集合中, 然后逐项加以检验。

首先是扫描。扫描一开始, 第一个操作是 getDiskReport(), 然后才是基于其结果的其他处理, 所以我们先看这个 getDiskReport() 的摘要:

```
[DirectoryScanner.run() > reconcile() > scan() > getDiskReport()]
```

```
DirectoryScanner.getDiskReport()
```

```

> List<?extends FsVolumeSpi> volumes = dataset.getVolumes()
> ScanInfoPerBlockPool[] dirReports = new ScanInfoPerBlockPool[volumes.size()]
> for (int i=0; i < volumes.size(); i++) { //为每个文件卷创建一个 ReportCompiler 线程
>+ if (isValid(dataset, volumes.get(i))) {
>++ reportCompiler = new ReportCompiler(volumes.get(i)) //这是个 Runnable
>++ Future<ScanInfoPerBlockPool>

```

```
    result = reportCompileThreadPool.submit(reportCompiler)
```

```
    //将此 Runnable 提交给 Java 虚拟机, 要求安排线程加以执行
```



```

> ++ compilersInProgress.put(i, result)
> + }
> }
> for (Entry<Integer, Future<ScanInfoPerBlockPool>> report : compilersInProgress.entrySet()) {
    // 对于所提交的各个线程,即各个文件卷
> + dirReports[report.getKey()] = report.getValue().get()
    // 收集其运行结果于 dirReports[], 这是一个文件卷的 ScanInfoPerBlockPool
> } // end for
> // Compile consolidated report for all the volumes
> ScanInfoPerBlockPool list = new ScanInfoPerBlockPool() // 用于汇总
> for (int i = 0; i < volumes.size(); i++) { // 对于本节点上的各个文件卷
> + if (isValid(dataset, volumes.get(i))) { // volume is still valid, 如果文件卷仍有效
> ++ list.addAll(dirReports[i]) // 就将相应的数组元素加入汇总的 ScanInfoPerBlockPool
> }
> return list.toSortedArrays() // 返回汇总的结果, 这是整个节点的 ScanInfoPerBlockPool

```

由一个 NameNode 存储在某个 DataNode 上的数据块, 可能存放在若干不同的文件卷中。而一个文件卷中也可以有属于不同 BlockPool 的子目录。ReportCompiler 线程是为文件卷创建的, 一个 ReportCompiler 线程负责提供一个文件卷的报告。不过 ReportCompiler 是 Runnable, 所以得把它们提交给 JVM 由其安排线程加以执行, 执行时就调用 reportCompiler 提供的 call() 函数, 有许多操作是在这个 call() 函数中完成的。

```

ReportCompiler.call()
> String[] bpList = volume.getBlockPoolList() // 本文件卷上可以有多个 BlockPool
> result = new ScanInfoPerBlockPool(bpList.length) // 创建一个 ScanInfo 链表的集合
> for (String bpid : bpList) { // 对于本文件卷上的各个 BlockPool
> + LinkedList<ScanInfo> report = new LinkedList<ScanInfo>() // 创建一个 ScanInfo 链表
> + File bpFinalizedDir = volume.getFinalizedDir(bpid) // 获取该 BlockPool 的 finalized 目录
> + rpt = compileReport(volume, bpFinalizedDir, report) // 生成其 ScanInfo 链表的内容
> + result.put(bpid, rpt) // 将此 ScanInfo 链表放在链表集合中
> }
> return result // 返回本文件卷所支持的各 BlockPool 部分的整个 ScanInfo 链表集合

```

显然, 这个线程为本文件卷上的每个 BP 子目录都编制一个报告。所返回的结果 result 是个 ScanInfo 链表的集合, 即链表的链表。文件卷中有几个 BP 的子目录, 链表 result 中就有几个元素, 各自代表着一个 BP 子目录。然后, 在代表着具体 BP 子目录的链表中, 则目录中有多少个块文件就会有多少个元素。关于每个 BP 子目录的报告由 compileReport() 完成:

```

[ReportCompiler.call() > compileReport()]

```

```

LinkedList<ScanInfo> compileReport(FsVolumeSpi vol, File dir, LinkedList<ScanInfo> report)
> File[] files = FileUtil.listFiles(dir) // 对该目录执行列表操作
> Arrays.sort(files) // 对输出进行排序

```

```

> for (int i = 0; i < files.length; i++) { //对于每个目录项
>+ if (files[i].isDirectory()) { //如果是子目录
>++ compileReport(vol, files[i], report) //递归
>++ continue
>+ }
>+ if (!Block.isBlockFilename(files[i])) { //如果文件名不是块文件的文件名
>++ if (isBlockMetaFile("blk_", files[i].getName())) { //但却是元文件的文件名
>+++ long blockId = Block.getBlockId(files[i].getName()) //从文件名中抽取 BlockId
>+++ report.add(new ScanInfo(blockId, null, files[i], vol)) //创建 ScanInfo 并加入报告
>++ }
>++ continue
>+ }
>+ //文件名是块文件的文件名
>+ File blockFile = files[i]
>+ long blockId = Block.filename2id(blockFile.getName()) //从文件名中抽取 BlockId
>+ while (i + 1 < files.length && files[i + 1].isFile()
        && files[i + 1].getName().startsWith(blockFile.getName())) {
>++ i++
>++ if (isBlockMetaFile(blockFile.getName(), files[i].getName())) {
>+++ metaFile = files[i]
>+++ break
>++ }
>++ report.add(new ScanInfo(blockId, blockFile, metaFile, vol)) //创建 ScanInfo 并加入报告
>+ }
>+ return report
> }

```

这样,当所有的 ReportCompiler 线程结束运行时,我们就有了一组“链表的链表”、一组 ScanInfoPerBlockPool,再在 getDiskReport() 中加以汇总。于是当程序回到 scan() 中时, diskReport 就是这份汇总。有了这份汇总的块文件清单,就可以与保存在内存中的 Block 清单进行比对而产生差异记录 diffRecord 了:

```

[DirectoryScanner.run() >reconcile()> scan()]

DirectoryScanner.scan()
> clear()
> Map<String, ScanInfo[]> diskReport = getDiskReport() //diskReport 是个链表的链表
> for (Entry<String, ScanInfo[]> entry : diskReport.entrySet()) { //对文件卷中的每个 BP 目录
>+ String bpid = entry.getKey()
>+ ScanInfo[] blockpoolReport = entry.getValue()
>+ Stats statsRecord = new Stats(bpid)

```

```

>+ stats.put(bpid, statsRecord)
>+ LinkedList<ScanInfo> diffRecord = new LinkedList<ScanInfo>()
    //为对应于具体 BlockPool 的 BP 目录创建一个空白的 diffRecord
>+ diffs.put(bpid, diffRecord) //把这个 diffRecord 及其 bpid 加入 diffs 集合中
>+ statsRecord.totalBlocks = blockpoolReport.length
>+ List<FinalizedReplica> bl = dataset.getFinalizedBlocks(bpid)
    //从文件卷的 volumeMap 中获取该 BP 目录下所有已 Finalized 的数据块文件清单
>+ FinalizedReplica[] memReport = bl.toArray(new FinalizedReplica[bl.size()])
    //获取 BlockPool 中有记载的数据块备份清单
>+ Arrays.sort(memReport); // Sort based on blockId, 按块号排序
>+ m = 0; // index for memReport
>+ while (m < memReport.length && d < blockpoolReport.length) {
    //逐项比对 memReport 与 blockpoolReport, 二者都已按 BlockId 排序
>++ FinalizedReplica memBlock = memReport[Math.min(m, memReport.length - 1)]
>++ ScanInfo info = blockpoolReport[Math.min(d, blockpoolReport.length - 1)]
>++ if (info.getBlockId() < memBlock.getBlockId()) { // Block is missing in memory
    // memReport 中缺了一个 BlockId, 文件存在但没有相应的记载
>+++ statsRecord.missingMemoryBlocks ++
>+++ addDifference(diffRecord, statsRecord, info)
>+++ d++
>+++ continue
>++ }
>++ if (info.getBlockId() > memBlock.getBlockId()) { // Block is missing on the disk
    //blockpoolReport 中缺了一个 BlockId, 文件缺失
>+++ addDifference(diffRecord, statsRecord, memBlock.getBlockId(), info.getVolume())
>+++ m++
>+++ continue
>++ }
>++ // Block file and/or metadata file exists on the disk. Block exists in memory.
>++ if (info.getBlockFile() == null) { // Block metadata file exists and block file is missing
    //元文件存在但数据文件缺失
>+++ addDifference(diffRecord, statsRecord, info)
>++ } else if (info.getGenStamp() != memBlock.getGenerationStamp() /* 世代标记不符 */
    || info.getBlockFileLength() != memBlock.getNumBytes() /* 长度不符 */ {
>+++ // Block metadata file is missing or has wrong generation stamp,
>+++ // or block file length is different than expected
>+++ statsRecord.mismatchBlocks ++
>+++ addDifference(diffRecord, statsRecord, info)
>++ } else if (info.getBlockFile().compareTo(memBlock.getBlockFile()) != 0) {
>+++ // volumeMap record and on-disk files don't match

```

```

>++++ statsRecord.duplicateBlocks ++
>++++ addDifference(diffRecord, statsRecord, info)
>+++ }
>+++ d++
>+++ if (d < blockpoolReport.length) { blockpoolReport 尚未到达最后一项
>++++ // There may be multiple on-disk records for the same block,
>++++ // don't increment the memory record pointer if so.
>++++ ScanInfo nextInfo = blockpoolReport[Math.min(d, blockpoolReport.length - 1)]
>++++ if (nextInfo.getBlockId() != info.blockId) ++m
           //如果 blockpoolReport 中的下一项属于同一 BlockId,就不推进 memReport
>+++ } else ++m // blockpoolReport 已到达最后一项,memReport 向前推进
>+ } //end while
>+ //至此,memReport 与 blockpoolReport 至少有一方已经穷尽,逐项比对完毕
>+ while (m < memReport.length) { //如果 memReport 中尚有剩余
>++ FinalizedReplica current = memReport[m++]
>++ addDifference(diffRecord, statsRecord, current.getBlockId(), current.getVolume())
>+ }
>+ while (d < blockpoolReport.length) { //如果 blockpoolReport 中尚有剩余
>++ statsRecord.missingMemoryBlocks ++
>++ addDifference(diffRecord, statsRecord, blockpoolReport[d++])
>+ }
> } //end for

```

如前所述,DataNode 上对每个 FSNamesystem 即每个 NameNode 都有个 BlockPool,实际上是 BlockPoolSlice,里面记录着这个 NameNode 存放在本节点上的所有数据块复份,即数据块文件。所以,BlockPool 就像账本,而文件卷就像仓库,前者在内存中,后者在磁盘上或别的存储介质上。然而“账”与“实”可能不符,BlockPool 中的信息可能与实际存储着的块文件有所不符。特别地,在 DataNode 启动之初这个 BlockPool 的内容根本就是空白。所以就要通过扫描来发现双方的差异,再用此差异来调整 BlockPool 的内容,这就好像一次仓库盘点。不过这个 scan() 函数并非只针对一个 BlockPool,而是针对一个文件卷。如我们在前面所见,由于支持联邦模式,一个文件卷上可以有多个 BP 目录。

程序中,每当完成了内层的 while 循环,并在循环外面把一方已经穷尽后另一方的剩余全都加入 diffRecord,就完成了针对一个 BlockPool 的比对,所有的差异都在 diffRecord 中。而这个 diffRecord,则是一个 ScanInfo 对象的列表(List),每个 ScanInfo 代表着一个账实不符、双方存在差异的数据块(复份)或数据块文件。至于 diffs,则是对所有 BlockPool 的汇总,这是个 ScanInfoPerBlockPool 对象的 Map,相当于一个便查表,使用具体 BlockPool 的 bpid 就可从中找到其 diffRecord。

到了完成外层的 for 循环时,这个文件卷上所有的 BP 子目录都已跟内存中 BlockPool 的信息经过了比对,集合 diffs 中汇集了所有的差异。

所以,当程序从 scan() 返回到 reconcile() 中时,所有的差异都在 diffs 中。

可以想见,一个 DataNode 的初始化刚完成时,内存中记录的数据块集合是空,而持久存储在磁盘上的每个块文件都会被看成差异,从而都在 diffs 集合中,此时的 diffs 实际上就是一份本节点上的块文件清单。

回到 reconcile() 中之后,是一个二层嵌套的 for 循环,对 diffs 中的每一项差异进行 checkAndUpdate(),以确认差异的真实性,并对内存中的数据块集合进行相应的修正:

```
[DirectoryScanner.run() > reconcile() > FsDatasetImpl.checkAndUpdate()]
```

```
FsDatasetImpl.checkAndUpdate(String bpid, long blockId, File diskFile,
                                File diskMetaFile, FsVolumeSpi vol)
    //对记录于 BlockPool 集合中的一项具体的差异进行验证
> ReplicaInfo memBlockInfo = volumeMap.get(bpid, blockId)
> if (memBlockInfo != null && memBlockInfo.getState() != ReplicaState.FINALIZED) return
    //尚未 Finalize,不足为凭,暂予忽略
> diskGS = diskMetaFile != null && diskMetaFile.exists()?
    Block.getGenerationStamp(diskMetaFile.getName()) :
    GenerationStamp.GRANDFATHER_GENERATION_STAMP
    //GS 是 Generation stamp 的缩写,类似于世代编号
> if (diskFile == null || !diskFile.exists()) { //处理磁盘上文件缺失的情况
>+ if (memBlockInfo == null) {
>++ if (diskMetaFile != null && diskMetaFile.exists() && diskMetaFile.delete()) {
>+++ LOG.warn("Deleted a metadata file without a block " + diskMetaFile.getAbsolutePath())
>+++ }
>++ return
>+ }
>+ if (!memBlockInfo.getBlockFile().exists()) {
>++ volumeMap.remove(bpid, blockId)
    //Block is in memory and not on the disk, Remove the block from volumeMap
>++ DataBlockScanner blockScanner = datanode.getBlockScanner()
>++ if (blockScanner != null) blockScanner.deleteBlock(bpid, new Block(blockId))
>++ if (vol.isTransientStorage()) {
>+++ ramDiskReplicaTracker.discardReplica(bpid, blockId, true)
>+++ }
>++ LOG.warn("Removed block " + blockId +
    " from memory with missing block file on the disk")
>++ // Finally remove the metadata file
>++ if (diskMetaFile != null && diskMetaFile.exists() && diskMetaFile.delete()) {
>+++ LOG.warn("Deleted a metadata file for the deleted block " +
    diskMetaFile.getAbsolutePath())
>+++ }
>+ }
```

```

>+ return
> } //end if (diskFile == null || !diskFile.exists())
> //Block file exists on the disk
> if (memBlockInfo == null) {    //处理内存中信息缺失的情况
    // Block is missing in memory - add the block to volumeMap
>+ ReplicaInfo diskBlockInfo = new FinalizedReplica(blockId, diskFile.length(),
    diskGS, vol, diskFile.getParentFile())

>+ volumeMap.add(bpid, diskBlockInfo)
>+ DataBlockScanner blockScanner = datanode.getBlockScanner()
>+ if (!vol.isTransientStorage()) {
>++ if (blockScanner != null) blockScanner.addBlock(new ExtendedBlock(bpid, diskBlockInfo))
>+ } else {
>++ ramDiskReplicaTracker.addReplica(bpid, blockId, (FsVolumeImpl) vol)
>+ }
>+ LOG.warn("Added missing block to memory " + diskBlockInfo)
>+ return
> }

> //Block exists in volumeMap and the block file exists on the disk. Compare block files
> File memFile = memBlockInfo.getBlockFile()
> if (memFile.exists()) {    //内存文件存在:
>+ if (memFile.compareTo(diskFile) != 0) {
>++ if (diskMetaFile.exists()) {
>+++ if (memBlockInfo.getMetaFile().exists()) {
>++++ // We have two sets of block + meta files. Decide which one to keep.
>+++++ ReplicaInfo diskBlockInfo = new FinalizedReplica(blockId, diskFile.length(),
    diskGS, vol, diskFile.getParentFile())
>+++++ ReplicaInfo rep = ((FsVolumeImpl) vol).getBlockPoolSlice(bpid)
>+++++ rep.resolveDuplicateReplicas(memBlockInfo, diskBlockInfo, volumeMap)
>++++ }
>++ } else {    //!diskMetaFile.exists()
>+++ diskFile.delete()    //磁盘上只有块文件,没有元数据文件,就删除块文件
>++ }
>+ } //end if (memFile.compareTo(diskFile) != 0)
> } else {    //!memFile.exists(),内存文件不存在(但磁盘上有):不要紧,加上就行
>+ // Block refers to a block file that does not exist.
>+ LOG.warn("Block file in volumeMap " + memFile.getAbsolutePath()
    + " does not exist. Updating it to the file found during scan "
    + diskFile.getAbsolutePath())
>+ memBlockInfo.setDir(diskFile.getParentFile())
>+ memFile = diskFile

```



```

>+ LOG.warn("Updating generation stamp for block " + blockId +
            " from " + memBlockInfo.getGenerationStamp() + " to " + diskGS)
>+ memBlockInfo.setGenerationStamp(diskGS)
> }
> // Compare generation stamp
> if (memBlockInfo.getGenerationStamp() != diskGS) { //世代标记不符
>+ File memMetaFile = FsDatasetUtil.getMetaFile(diskFile,
                                                memBlockInfo.getGenerationStamp())
>+ if (memMetaFile.exists()) {
>++ if (memMetaFile.compareTo(diskMetaFile) != 0) {
>+++ LOG.warn("Metadata file in memory " + memMetaFile.getAbsolutePath()
            + " does not match file found by scan "
            + (diskMetaFile == null ? null : diskMetaFile.getAbsolutePath()))
>++ }
>+ } else {
>++ // Metadata file corresponding to block in memory is missing
>++ gs = diskMetaFile != null && diskMetaFile.exists() &&
            diskMetaFile.getParent().equals(memFile.getParent()) ? diskGS :
            GenerationStamp.GRANDFATHER_GENERATION_STAMP
>++ LOG.warn("Updating generation stamp for block " + blockId +
            " from " + memBlockInfo.getGenerationStamp() + " to " + gs)
>++ memBlockInfo.setGenerationStamp(gs)
>+ }
> }
> // Compare block size
> if (memBlockInfo.getNumBytes() != memFile.length()) { //大小不符
>+ // Update the length based on the block file
>+ corruptBlock = new Block(memBlockInfo)
>+ LOG.warn("Updating size of block " + blockId + " from "
            + memBlockInfo.getNumBytes() + " to " + memFile.length())
>+ memBlockInfo.setNumBytes(memFile.length())
> }
> // Send corrupt block report outside the lock
> if (corruptBlock != null) { //存在坏块
>+ LOG.warn("Reporting the block " + corruptBlock + " as corrupt due to length mismatch")
>+ datanode.reportBadBlocks(new ExtendedBlock(bpid, corruptBlock))
//向 DataNode 提交坏块报告,最终会报告给 NameNode
> }

```

目录扫描的产出是内容经过更新的 BlockPool 和可能存在的坏块报告,这都将成为向 NameNode 报告的主要内容。

12.5 数据块扫描线程 DataBlockScanner

DirectoryScanner 的作用类似于清点仓库,搞清本地存储着一些什么数据块(复份),但并不确定所存储的数据块文件仍旧完好无损。所以 HDFS 中还有一个 DataBlockScanner,它的作用是周期地轮流读一遍本地的数据文件,读的过程中加以 CRC 之类的校验,如果能顺利读出而不发生异常,就说明数据文件完好。

```
class DataBlockScanner implements Runnable {}
] DataNode datanode
] FsDatasetSpi<? extends FsVolumeSpi> dataset
] run()
    > String currentBpId = "" //空,最初时还不知道该扫描哪一个 BlockPoolSlice
    > while (datanode.shouldRun &&!Thread.interrupted()) {
    >+ if (!firstRun) Thread.sleep(SLEEP_PERIOD_MS)
    >+ BlockPoolSliceScanner bpScanner = getNextBPScanner(currentBpId)
    >+ currentBpId = bpScanner.getBlockPoolId() //获取其 bpid
    >+ bpScanner.scanBlockPoolSlice() == BlockPoolSliceScanner.scanBlockPoolSlice()
    > } //end while
    > for (BlockPoolSliceScanner bpss: blockPoolScannerMap.values()) {
    >+ bpss.shutdown()
    > }
```

这是一个线程,每睡眠一段时间以后就来活动一下,每次活动都扫描一个 BlockPoolSlice。我们知道,一个 FSNamesystem(实际上就是一个 NameNode)对应着一个 BlockPool,这是它所有数据块的集合,但是这个集合分布在集群内的众多 DataNode 上,每个 DataNode 上所存储的只是这个 BlockPool 的一个“切片”,即 BlockPoolSlice。所以,BlockPool 只是概念上的存在,BlockPoolSlice 才是实际的存在。

每个 DataNode 上都有一个 DataBlockScanner,当然它只能扫描存在于本节点上的那一部分。但是现在的 HDFS 支持联邦模式,一个集群中可以有多个 NameNode、多个 BlockPool,于是在一个 DataNode 上就可以有多个 BlockPoolSlice,得要分别加以扫描。这个线程的主循环每一轮都只扫描一个 BlockPoolSlice,但是对具体 BlockPoolSlice 的扫描需要集中和模块化,因为对一个 BlockPoolSlice 的扫描并非一次就能完成,中间会产生很多数据,这些数据下次扫描时还要用到。所以 Hadoop 的代码中就安排了一个 BlockPoolSliceScanner 类,为每个 BlockPoolSlice 都配上专职的 BlockPoolSliceScanner,对具体 BlockPoolSlice 的扫描都是由 BlockPoolSliceScanner 进行的。

当然,首先要确定扫描哪一个 BlockPoolSlice,并找到它的 BlockPoolSliceScanner 对象,这就是为什么先要调用 getNextBPScanner():

```
[DataBlockScanner.run() > getNextBPScanner()]
```

```

getNextBPScanner(String currentBpId)
> while (datanode.shouldRun &&!blockScannerThread.isInterrupted()) {
>+   waitForInit()
>+   if (getBlockPoolSetSize() > 0) { //BlockPoolSliceScanner 的集合非空
>++   lastScanTime = 0
>++   for (String bpid : blockPoolScannerMap.keySet()) {
//在这个 MAP 中寻找最长时间没有得到扫描的 BlockPoolSlice;
>++++   long t = getBPScanner(bpid).getLastScanTime()
>++++   if (t != 0L) {
>+++++   if (bpid == null || t < lastScanTime) {
>+++++   lastScanTime = t
>+++++   nextBpId = bpid
>+++++   }
>++++   }
>++   } //end for
>++   if (nextBpId == null) { //要是找不到,就
>++++   nextBpId = blockPoolScannerMap.higherKey(currentBpId)
//从 MAP 中找大于 currentBpId 的最小 bpid
>++++   if (nextBpId == null) nextBpId = blockPoolScannerMap.firstKey()
//还没有,就用 MAP 中的第一项
>++   }
//有了 bpid,就可得到其 BlockPoolSliceScanner
>++   if (nextBpId != null) return getBPScanner(nextBpId)
>++> return blockPoolScannerMap.get(bpid) //从 MAP 中按 bpid 找到 BlockPoolSliceScanner
>+ } //end if (getBlockPoolSetSize() > 0)
>+ Thread.sleep(5000)
> } //end while
> return null

```

节点上有几个 BlockPoolSlice,就有几个 BlockPoolSliceScanner,blockPoolScannerMap 则是这些扫描器的集合。但是选谁呢?每次都是选其中最长时间没有得到扫描的 BlockPoolSlice,如果都不相上下再用别的办法。显然,这是因为假定集群中有多个 NameNode,因而 DataNode 上有多个 BlockPoolSlice 才需要这样,但是实际上绝大多数的 Hadoop 集群都只有一个 NameNode。

既然实际的扫描是由 BlockPoolSliceScanner 进行的,我们就也要看一下它的摘要:

```

class BlockPoolSliceScanner {
] int MAX_SCAN_RATE = 8 * 1024 * 1024; //最大每秒扫描 8MB
] int MIN_SCAN_RATE = 1 * 1024 * 1024; //最小每秒扫描 1MB
] long DEFAULT_SCAN_PERIOD_HOURS = 21 * 24L; // three weeks
] String VERIFICATION_PREFIX = "dncp_block_verification.log"
] String blockPoolId

```

```

] long scanPeriod
] FsDatasetSpi<?extends FsVolumeSpi> dataset
] SortedSet<BlockScanInfo> blockInfoSet           //按时间排序的 BlockScanInfo 对象集合
] SortedSet<BlockScanInfo> newBlockInfoSet
] GSet<Block, BlockScanInfo> blockMap
] HashMap<Long, Integer> processedBlocks
- - - - - 以上为数据结构部分 - - - - -
] addBlock(ExtendedBlock block)
           //将 block 加入 blockMap,并视情况加入 blockInfoSet 或 newBlockInfoSet
] adjustThrottler()
  > throttler.setBandwidth(Math.min(bw, MAX_SCAN_RATE))
] scanBlockPoolSlice()
] scan()

```

确定了哪一个 BlockPoolSliceScanner 之后,就调用它的 scanBlockPoolSlice()加以扫描:

```
[DataBlockScanner.run() > BlockPoolSliceScanner.scanBlockPoolSlice()]
```

```
BlockPoolSliceScanner.scanBlockPoolSlice()
```

```

> if (!workRemainingInCurrentPeriod()) return //如果无事可干(例如时间未到)就直接返回
> processedBlocks = new HashMap<Long, Integer>() //创建 processedBlocks 用于临时记录
> b = assignInitialVerificationTimes() // Reads the current and previous log files (if any) and
           //marks the blocks processed if they were processed within last scan period.
           //Copies the log records of recently scanned blocks from previous to current file.
           //Returns false if the process was interrupted because the thread is marked to exit.
> if (!b) return //准备工作未能顺利完成,不能进行扫描,返回
> scan() == BlockPoolSliceScanner.scan()
> totalBlocksScannedInLastRun.set(processedBlocks.size())
           //本次扫描中成功通过校验的块(复份)都在 processedBlocks 集合中
> lastScanTime.set(Time.monotonicNow())

```

这里,assignInitialVerificationTimes()是做准备,scan()是实际的扫描验证。

一个 BlockPoolSlice 的内容可能是不断在变化的,所以对 BlockPoolSlice 的扫描并不简单,它要检查 Log 文件,看上次扫描之后有了些什么变化,新增了哪一些数据块复份。

我们先看这准备工作是怎么做的。

```

[DataBlockScanner.run() > BlockPoolSliceScanner.scanBlockPoolSlice()
> assignInitialVerificationTimes()]

```

```
BlockPoolSliceScanner.assignInitialVerificationTimes()
```

```

> //First updates the last verification times from the log file.
> if (verificationLog!= null) {           // LogFileHandler 已创建
>+ long now = Time.monotonicNow()

```

```

>+ RollingLogs.LineIterator logIterator = null
    //实际是 RollingLogsImpl 对象,实现.curr 和.prev 两个 Log 文件,交替滚进
>+ logIterator = verificationLog.logs.iterator(false)
    //准备逐项扫描 Log 文件的内容,不跳过.prev 文件,把两个 Log 文件串在一起
>+// update verification times from the verificationLog.
>+ while (logIterator.hasNext()) { //只要 Log 文件中还有记录项
>++ LogEntry entry = LogEntry.parseEntry(logIterator.next()) //将一行记录转成一个 LogEntry
>++ if (entry!= null) {
>+++ updateBlockInfo(entry) //根据该记录的生成时间和该数据块上次的验证时间来设置
>++++> BlockScanInfo info = blockMap.get(new Block(e.blockId, 0, e.genStamp))
    //从 blockMap 中找到该 Log 记录所指数据块的 BlockScanInfo
>++++> if(info!= null && e.verificationTime > 0 && info.lastScanTime < e.verificationTime) {
    //如果确认该记录有效,并且产生于上次扫描之后,则该数据块已有变化
>++++>+ delBlockInfo(info) //先将此 BlockScanInfo 从 blockMap 脱钩
>++++>+ info.lastScanTime = e.verificationTime //将其 lastScanTime 改成 Log 所记录的时间
>++++>+ info.lastScanType = ScanType.VERIFICATION_SCAN //设置其扫描类型
>++++>+ addBlockInfo(info, false) //挂回 blockMap
>++++> } //end if, end updateBlockInfo()
>++++ if (now - entry.verificationTime < scanPeriod) { //如果时间间隔尚未到点
>+++++ BlockScanInfo info = blockMap.get(new Block(entry.blockId, 0, entry.genStamp))
    //再从 blockMap 中找回其 BlockScanInfo(位置已有改变)
>+++++ if (info!= null) {
>++++++ if (processedBlocks.get(entry.blockId) == null) { //从 processedBlocks 中取出
>>+++++++ if (isNewPeriod) updateBytesLeft(- info.getNumBytes())
>>+++++++> bytesLeft += len //其实是从 BlockPoolSliceScanner.bytesLeft 上减去本块大小
>>+++++++ processedBlocks.put(entry.blockId, 1) //放回 processedBlocks 中
>++++++ }
>++++++ if (logIterator.isLastReadFromPrevious()) { //如果 Log 记录读自先前 Log 文件
>>+++++++ //write the log entry to current file so that the entry is preserved for later runs.
>>+++++++ verificationLog.append(entry.verificationTime, entry.genStamp, entry.blockId)
    //就将其写入当前 Log 文件,以防因滚进而丢失
>++++++ }
>+++++ } //end if (info!= null)
>+++ } //end if (now - entry.verificationTime < scanPeriod)
>++ } //end if (entry!= null)
>+ } //end while
>+ isNewPeriod = false
> } //end if (verificationLog!= null)
-----
> //Before this loop, entries in blockInfoSet that are not updated above

```

```

                                have lastScanTime of <= 0
> //Loop until first entry has lastModificationTime > 0.
> int numBlocks = Math.max(blockMap.size(), 1)
> // Initially spread the block reads over half of scan period so that
                                we don't keep scanning the blocks too quickly when restarted.
> verifyInterval = Math.min(scanPeriod/(2L * numBlocks), 10 * 60 * 1000L)
> lastScanTime = Time.monotonicNow() - scanPeriod
> if (!blockInfoSet.isEmpty()) { // blockInfoSet 非空
>+ BlockScanInfo info
>+ while ((info = blockInfoSet.first()).lastScanTime < 0) {
                                //排序集合 blockInfoSet 中所有元素是按 lastScanTime 排序的
>++ delBlockInfo(info) //从排序集合 blockInfoSet 中删去 lastScanTime 为负的元素
>++ info.lastScanTime = lastScanTime //改变其 lastScanTime
>++ lastScanTime += verifyInterval //推进最后扫描时间
>++ addBlockInfo(info, false) //将此 BlockScanInfo 加回排序集合中
>+ } //end while
> }
> return true //如果没发生异常,顺利到达终点,就完成了准备;否则会返回 false

```

准备完了以后,就是实际的扫描校验了。所谓校验,就是以此将 blockInfoSet 集合中的每个数据块复份在本地文件系统中的块文件读出一遍,并在读出过程中进行类似于 CRC 那样的校验,如果通不过校验就会失败。而所谓“读出”,当然不能把整个文件的内容都读到缓冲区中,而是边读边扔,办法就是把读入的内容“发送”到一个类似于/dev/null 那样的 NullOutputStream 中。对于成功通过了校验的块,就把其 BlockScanInfo 对象从 blockInfoSet 集合转移到 processedBlocks 中。

```
[DataBlockScanner.run() > BlockPoolSliceScanner.scanBlockPoolSlice() > scan()]
```

```

BlockPoolSliceScanner.scan()
> adjustThrottler() //调整发送的速度
>> throttler.setBandwidth(Math.min(bw, MAX_SCAN_RATE))
> while (datanode.shouldRun && ...) { //只要一切正常
>+ if (now >= (currentPeriodStart + scanPeriod)) { //已经超出上一个周期的时间
>++ startNewPeriod() //开始一个新的周期
>++> bytesLeft = totalBytesToScan //本周期中需要扫描验证的数据长度
>++> currentPeriodStart = Time.monotonicNow() //以当前时间为本周期的起点
>++> isNewPeriod = true
>+ }
>+ st = getEarliestScanTime()
>+> if (!blockInfoSet.isEmpty()) {
>+>+ return blockInfoSet.first().lastScanTime;

```



```

>+> }
>+> return Long.MAX_VALUE
>+ if (((now - st) >= scanPeriod)
        || ((!blockInfoSet.isEmpty()) &&!(this.isFirstBlockProcessed()))){
>++ verifyFirstBlock() // blockInfoSet 是个 SortedSet,即排序集合
>+ } else {
>++ LOG.debug("All remaining blocks were processed recently, " + "so this run is complete")
>++ break //已经没有数据块需要处理了,跳出 while 循环
>+ }
> } //end while
-----
> rollNewBlocksInfo() //把 newBlockInfoSet 中的内容转移到 blockInfoSet 中
>> for (BlockScanInfo newBlock : newBlockInfoSet) {
>>+ blockInfoSet.add(newBlock)
>> }
>> newBlockInfoSet.clear() //释放 newBlockInfoSet,下次扫描时将另行创建

```

这个 while 循环实际上是对 blockInfoSet 的循环,这是个按时间排序的集合,只要该集合非空,就总有一个元素成为 FirstBlock,如果这个 FirstBlock 符合条件就调用 verifyFirstBlock(),否则就通过 break 语句结束循环。这里的 getEarliestScanTime()实际上就是取 FirstBlock 的 lastScanTime,即最近一次受到扫描的时间;而 isFirstBlockProcessed()则检查 FirstBlock 的块号 blockId 是否在集合 processedBlocks 中。因为是经过排序的 SortedSet,所以如果其 FirstBlock 不满足条件,那么此后的所有 Block 就都不会满足条件了。

对于满足条件的 Block,要做的事情是校验,即 verify,这就是 DataBlockScanner 要达到的目标。

```

[DataBlockScanner.run() > BlockPoolSliceScanner.scanBlockPoolSlice() > scan() >
verifyFirstBlock()]

```

```

BlockPoolSliceScanner.verifyFirstBlock()
> BlockScanInfo block = null
> if (!blockInfoSet.isEmpty()) block = blockInfoSet.first() //从 blockInfoSet 中取出第一个块
> if( block!= null ) {
>+ verifyBlock(new ExtendedBlock(blockPoolId, block))
>+> for (int i = 0; i < 2; i++) { //如果第一次验证失败,就再来一次,但不超过两次
>+>+ boolean second = (i > 0)
>+>+ adjustThrottler() //酌情调整发送的速度
>+>+ try{
>+>+ blockSender = new BlockSender(block, 0, -1, false, true, true, ...,
                                CachingStrategy.newDropBehind())
                                // BlockSender 的输入来自 BlockScanInfo 所指的块文件

```

```

>+>+ out = new DataOutputStream(new IOUtils.NullOutputStream())
//BlockSender 的输出是 NullOutputStream
>+>+ blockSender.sendBlock(out, null, throttler)
    == BlockSender.sendBlock(out, null, throttler)
//从块文件读入,输出到 NullOutputStream,若读入出错就会发生异常
>+>+ LOG.info((second?"Second " : "") + "Verification succeeded for " + block)
>+>+ if( second ) totalTransientErrors++ //如果是第二次验证
>+>+ updateScanStatus((BlockScanInfo)block.getLocalBlock(),
    ScanType.VERIFICATION_SCAN, true)

>+>+ return
>+> } catch(IOException e){ //如果发生异常
>+>+ ...
>+>+ handleScanFailure(block) //扫到坏块时的处理
>+>+> datanode.reportBadBlocks(block) //报告坏块
>+>+ return
>+>+ }
>+>+ totalScans++
>+> } //end for, end verifyBlock()
----- 从 verifyBlock() 返回到 verifyFirstBlock() 之后 -----
>+ processedBlocks.put(block.getBlockId(), 1) //处理过了,放入 processedBlocks 集合
> } //end if( block!= null )

```

其实 verifyFirstBlock() 这一层是很简单的,就是从排序集合 blockInfoSet 中取出第一个 BlockScanInfo,并将其改建成一个 ExtendedBlock,然后就对其调用 verifyBlock(),实际的验证是由后者完成的。

这里就地展开了 verifyBlock() 的摘要,这个函数的代码得要仔细看了。

首先是创建了一个 BlockSender,顾名思义这应该是用来发送数据块的,果然下面就调用了它的 sendBlock()。我们看一下这个类的操作方法摘要(只是很小一部分)。

```

class BlockSender implements java.io.Closeable {
] BlockSender(ExtendedBlock block, long startOffset, long length, boolean corruptChecksumOk,
    boolean verifyChecksum, boolean sendChecksum, DataNode datanode,
    String clientTraceFmt, CachingStrategy cachingStrategy)
] verifyChecksum(final byte[] buf, final int dataOffset, final int datalen,
    final int numChunks, final int checksumOffset)
] sendBlock(DataOutputStream out, OutputStream baseStream, DataTransferThrottler throttler)
] manageOsCache() //Manage the OS buffer cache by performing read-ahead and drop-behind.
] sendPacket(ByteBuffer pkt, int maxChunks, OutputStream out,
    boolean transferTo, DataTransferThrottler throttler)

```

我们把构造方法 BlockSender() 的参数表与前面代码中创建这个对象时的实际参数进行比对,就可看出参数 verifyChecksum 的值是 true,说明需要对数据块进行验和计算。另一个

第13章

DataNode 与 NameNode 的互动

数据节点 DataNode 在运行中会与三种对端有互动。第一种是 NameNode, 如前所述, 对于数据块的存储地点, 虽然最初是由 NameNode 分配和指定的, 但相关的信息最终来自 DataNode 的报告。第二种是用户的 App(包括 Shell), 用户的 App 可以存在于集群内的任何节点上, 不过那是在独立的 JVM 上, 即使与 DataNode 同在一个节点上也互相独立; 然而真正把数据存储在 DataNode 上或从 DataNode 读取数据的却是 App(或 Shell)。第三种是集群中别的 DataNode, 就是说 DataNode 与 DataNode 之间也会有通信和互动, 这主要来自数据块复份 replica 的传输和转储。

数据块在 HDFS 文件系统中的存储是“狡兔三窟”的, 一个数据块要分别存储在若干不同的 DataNode 上, 但是系统并不要求 App 把一个数据块分别发送给几个 DataNode, 而只需发送给其中的一个, 后面就是 DataNode 之间的事了。

其实 DataNode 的跨节点交互还不止这三种, 例如 DataNode 在运行中也会产生日志信息, 如果采用专门的日志(注意运行日志不同于 EditLog)服务器, 那么这里也就会有跨节点的交互。又如作为 App 特例的 Web 服务器所在的节点与所有节点都有互动, 当然也包括 DataNode。但是, 主要还是上述的这三种互动, 理解了这三种互动, 对 HDFS 就有了基本的理解, 所以我们在本章中先考察 DataNode 与 NameNode 之间的互动。

13.1 DataNode 与 NameNode 的互动

DataNode 与 NameNode 之间基本的通信手段就是 RPC, 在此基础上, DataNode 与 NameNode 的互动大致上有三种。首先是登记(registration), DataNode 一经启动就应该主动与 NameNode 建立 RPC 连接, 并向其登记, 让 NameNode 知道有这么一个 DataNode 已经在位了。然后, 第二种互动是向 NameNode 发送心跳信号并在这上面搭载各种报告, 一来让它知道这个 DataNode 继续存在, 二来让它知道这个 DataNode 上的存储发生了一些什么变化, 特别是节点上有哪些数据块的复份, 以及还有多少资源可供使用, 这也总是由 DataNode 主动发起的。第三种互动, 则是 NameNode 发回响应信息, 由 DataNode 执行 NameNode 搭载在响应信息里的命令和要求。实际上这第二种和第三种方式是合在一起的, 一来一去构成一个回合, 只是二者要达到的目标不同。

这样, 在结构的、宏观的层面 DataNode 是服务的提供者, 是被动的一方, 而 NameNode 是要求服务的一方; 但是在具体操作的层面却总是由 DataNode 主动发起。

明白了这个大致的格局, 我们从 DataNode 的 startDataNode() 开始看 DataNode 与

NameNode 的互动,不过其中也涉及一些 DataNode 之间的互动。

```

DataNode.startDataNode()
> initDataXceiver(conf) //这是为了与 App 和别的 DataNode 互动而做的准备,暂略
> initIpcServer(conf)    //建立基于 ProtoBuf 的 RPC 层通信机制,
                        //这也是为 DataNode 间互动而做的准备
>>> InetAddress ipcAddr = NetUtils.createSocketAddr(
    conf.get(DFS_DATANODE_IPC_ADDRESS_KEY))
>>> RPC.setProtocolEngine(conf, ClientDatanodeProtocolPB.class, ProtobufRpcEngine.class)
>>> clientDatanodeProtocolXlator = new ClientDatanodeProtocolServerSideTranslatorPB(this)
>>> service = ClientDatanodeProtocolService
                        .newReflectiveBlockingService(clientDatanodeProtocolXlator)
>>> ipcServer = new RPC.Builder(conf).setProtocol(ClientDatanodeProtocolPB.class)
                        .setInstance(service).setBindAddress(ipcAddr.getHostName())...build()
>>> interDatanodeProtocolXlator = new InterDatanodeProtocolServerSideTranslatorPB(this)
>>> service = InterDatanodeProtocolService
                        .newReflectiveBlockingService(interDatanodeProtocolXlator)
>>> DFSUtil.addPBProtocol(conf, InterDatanodeProtocolPB.class, service, ipcServer)
>>> LOG.info("Opened IPC server at " + ipcServer.getListenerAddress())
-----
> blockPoolManager = new BlockPoolManager(this) //创建本地的 BlockPoolManager 对象
> blockPoolManager.refreshNamenodes(conf) //建立或重建与 NameNode 的连接
    == BlockPoolManager.refreshNamenodes(Configuration conf)
>>> Map<String, Map<String, InetAddress>>> newAddressMap =
    DFSUtil.getNNServiceRpcAddressesForCluster(conf)
    //获取集群中各 nameservice(可以不止一个)的地址
>>> doRefreshNamenodes(newAddressMap) //按名单中的 nameservice 地址逐一建立连接
>>>> for (String nameserviceId : addrMap.keySet()) { //对于 newAddressMap 中的每个地址
>>>>+ if (bpByNameserviceId.containsKey(nameserviceId)) {
>>>>++ Set<String> toRefresh.add(nameserviceId) //原来就有连接的是 toRefresh
>>>>+ } else {
>>>>++ Set<String> toAdd.add(nameserviceId) //原来没有连接的就是 toAdd
>>>>+ }
>>>> }
>>>> if (!toAdd.isEmpty()) { //Start new nameservices
>>>>+ LOG.info("Starting BPOfferServices for nameservices: " + ...)
>>>>+ for (String nsToAdd : toAdd) { //对于需要新建连接的每个 nameservice:
>>>>++ ArrayList<InetAddress> addrs =
    Lists.newArrayList(addrMap.get(nsToAdd).values())
>>>>++ BPOfferService bpos = createBPOS(addrs)
    //创建一个专门为其服务的 BPOfferService 对象

```

```

>>> ++> return new BPOfferService(nnAddrs, dn)
>>> ++>> for (InetSocketAddress addr : nnAddrs) { //对这 nameservice 的每个 NameNode
>>> ++>>> this.bpServices.add(new BPServiceActor(addr, this))
>>> ++>> } // end for (InetSocketAddress addr : nnAddrs)
>>> ++ bpByNameserviceId.put(nsToAddr, bpos) //这个 nameservice 现在有了 BPOfferService
>>> ++ offerServices.add(bpos)
>>> + } //end for (String nsToAddr : toAdd)
>>> } // end if
>>> startAll() //以具体用户的身份启动各个 BPServiceActor 对象:
>>>> act = new PrivilegedExceptionAction<Object>()
>>>>     ] run()
>>>>     > for (BPOfferService bpos : offerServices) {
>>>>     >+ bpos.start()
>>>>     > }
>>>> UserGroupInformation.getLoginUser().doAs(act)
>>>>> BPOfferService.run() //这就是上面动态定义的那个 run() 函数
>>>>>> BPOfferService.start()
>>>>>>> for (BPServiceActor actor : bpServices){
>>>>>>>> + actor.start() == BPServiceActor.start()
>>>>>>>> +> bpThread = new Thread(this, formatThreadName())
>>>>>>>>> //为每个 BPServiceActor 都创建一个线程
>>>>>>>> +> bpThread.setDaemon(true)
>>>>>>>> +> bpThread.start() //启动这个线程
>>>>>>>> }

```

我们在这里主要关心 BlockPoolManager.refreshNamenodes(), 前面 initDataXceiver() 是创建用于 DataNode 间传输数据块的通信机制, 而 initIpcServer() 是创建 DataNode 间 IPC/RPC 机制的服务端。在 DataNode 间的通信中, 每个 DataNode 都既可以是服务端又可以是客户端, 因而需要有“收发器”即 Xceiver 的功能。另外, DataNode 间互动所用的规程是 InterDatanodeProtocolPB。

代码摘要中已经加了注释, 但是还需补充一些说明。NameNode 代表着一种查名服务、即 nameservice, 通常一个集群中只有一种 nameservice, 但是如果把集群配置成联邦(federation)模式, 那就可以有多个 nameservice。DataNode 需要与集群中的每个 nameservice 都建立连接, 为它们提供数据存储服务, 因而为每个 nameservice 都要创建一个 BPOfferService 对象。然后, 由于 HA 即高可靠性的要求, 每个 nameservice 又可能会有多个 NameNode, 一般是主备各一。StandbyNN 是为 NameNode (ActiveNN) 提供热备的, 随时都有可能会接管 NameNode 的服务, 因而 DataNode 必须预先就建立好与 Standby 的连接, 不能到时候措手不及。所以在每个 BPOfferService 内部要为这个 nameservice 的每个 NameNode 都建立连接, 并为之创建一个 BPServiceActor 线程专门绑定于这个特定的 NameNode。总之, 对于集群中的每个 NameNode, 每个 DataNode 上都有一个 BPServiceActor 线程专门为其服务。但是作

为一个特例,如果集群既非联邦模式,也未开通 HA,那就只有一个 NameNode,DataNode 上也就只有一个 BPSERVICEActor 线程。

另外,所谓 BPOfferService 和 BPSERVICEActor,都是以 BP 开头的,这显然是“Block Pool”的缩写。说到底,DataNode 所提供的服务,无非就是作为“块池”的服务。

我们看一下 BPOfferService 的数据结构部分的摘要:

```
class BPOfferService {}
] NamespaceInfo bpNSInfo           //关于目标 Namespace 即 nameservice 的信息
] DatanodeRegistration bpRegistration //本节点向该 nameservice 的登记信息
] DataNode dn                      //所在的 DataNode
] BPSERVICEActor bpServiceToActive //通向当前活跃 NameNode 的 BPSERVICEActor
] List<BPSERVICEActor> bpServices //属于同一 nameservice 的所有 BPSERVICEActor
```

这里的 DatanodeRegistration,是具体的 BPSERVICEActor 代表 DataNode 向 NameNode 登记时双方议定的,以后每次通信交互都要校验这个信息。

我们不妨也看一下 BPSERVICEActor 的数据结构部分摘要,以加深理解:

```
class BPSERVICEActor implements Runnable {
] InetSocketAddress nnAddr //所连接的 NameNode 的地址
] BPOfferService bpos      //所属的 BPOfferService
] DatanodeProtocolClientSideTranslatorPB bpNamenode //NameNode 的 proxy
] RunningState runningState // CONNECTING, INIT_FAILED,
                             RUNNING, EXITED, FAILED
] Map<DatanodeStorage, PerStoragePendingIncrementalIBR> pendingIncrementalIBRperStorage
                             //有待向 NameNode 报告的存储变化,IBR 为增量块报告之意
] DataNode dn              //所在的 DataNode
] DatanodeRegistration bpRegistration //本节点向该 NameNode 的登记信息
```

创建了连接具体 NameNode 的 BPSERVICEActor 线程并启动后,这个线程便开始独立运行,进入它的 run()函数:

```
BPSERVICEActor.run()
> while (true) {
>+ connectToNNAndHandshake() //试图建立与 NameNode 的连接
>+ break //一旦与 NameNode 连上,就跳出这个循环,进入下一个 while 循环
> }
> while (shouldRun()) {
>+ offerService() //向 NameNode 发送心跳和报告,并执行 NameNode 发回的命令
> } // end while, in BPSERVICEActor.run()
```

这里有前后两个 while 循环,前面那个循环里有个 break 语句,只要程序能从 connectToNNAndHandshake()正常返回而不发生异常,就会通过 break 语句跳出循环。实际的代码中这个 connectToNNAndHandshake()是在 try{}中,所以如果连接失败就会睡眠一会之后再试,直至成功。后面那个循环却不一样,它就是反复调用 offerService(),这才是线程的主循环。

我们先看 `connectToNNAndHandshake()`, 连接到 NameNode 并握手:

```
[BPServiceActor.run() > connectToNNAndHandshake()]
```

```
BPServiceActor.connectToNNAndHandshake()
```

```
> bpNamenode = dn.connectToNN(nnAddr)    //get NN proxy
    //建立与 NameNode 的 RPC 连接, 所得 proxy 即为 bpNamenode
    //bpNamenode 的类型为 DatanodeProtocolClientSideTranslatorPB
> NamespaceInfo nsInfo = retrieveNamespaceInfo() //向 NameNode 索取版本信息
    //First phase of the handshake with NN - get the namespace info
>> while (shouldRun()) {
>>+ nsInfo = bpNamenode.versionRequest()    //这是对 NameNode 的 RPC 调用
>>+ break    //如果从 RPC 正常返回就跳出循环, 若发生异常就睡眠后再试
>>+ sleepAndLogInterrupts(5000, "requesting version info from NN")
>> }
>> checkNNVersion(nsInfo)
>> return nsInfo
- - - - - 已经连上 NameNode, 并得到 NameNode 发回的版本信息 - - - - -
> bpos.verifyAndSetNamespaceInfo(nsInfo)
>> if (this.bpNSInfo == null) { //说明这是第一次连上这个 nameservice
>>+ this.bpNSInfo = nsInfo
>>+ dn.initBlockPool(this) //准备好本节点为这个 nameservice 提供的 BlockPool
>>+> NamespaceInfo nsInfo = bpos.getNamespaceInfo()
>>+> setClusterId(nsInfo.clusterID, nsInfo.getBlockPoolID())
>>+> blockPoolManager.addBlockPool(bpos) //在 blockPoolManager 中增加一个 BlockPool
    //Register the new block pool with the BP manager
>>+>> bpByBlockPoolId.put(bpos.getBlockPoolId(), bpos)
>>+> initStorage(nsInfo) //准备好这个 Namespace 在本地的存储空间
>>+>> factory = FsDatasetSpi.Factory.getFactory(conf)
>>+>> checkDatanodeUuid()
>>+>> if (data == null) { //FsDatasetSpi<?extends FsVolumeSpi> data
>>+>>+ data = factory.newInstance(this, storage, conf) //创建 FsDatasetImpl 对象
>>+>> }
>>+> checkDiskError()
>>+> initPeriodicScanners(conf) //创建这个 BlockPool 的目录扫描线程和数据块扫描线程
>>+> data.addBlockPool(nsInfo.getBlockPoolID(), conf)
    == FsDatasetImpl.addBlockPool(nsInfo.getBlockPoolID(), conf)
>>+>> volumes.addBlockPool(bpid, conf) //添加属于这个 Namespace 的 BlockPool
>>+>>> File bpdire = new File(currentDir, bpid)
>>+>>> bp = new BlockPoolSlice(bpid, this, bpdire, conf) //但实际上是 BlockPoolSlice
>>+>>> bpSlices.put(bpid, bp)
```

```

>>+>> volumeMap.initBlockPool(bpid)
>>+>> volumes.getAllVolumesMap(bpid, volumeMap, ramDiskReplicaTracker)
>> } else { //这不是连上这个 Namespace 的第一个 BPSERVICEActor 线程,需要验证相符
>>+ checkNSEquality(bpNSInfo.getBlockPoolID(), nsInfo.getBlockPoolID(), "Blockpool ID")
>>+ checkNSEquality(bpNSInfo.getNamespaceID(),
                    nsInfo.getNamespaceID(), "Namespace ID")
>>+ checkNSEquality(bpNSInfo.getClusterID(), nsInfo.getClusterID(), "Cluster ID")
>> }
> register() //正式向 NameNode 登记
>> DatanodeRegistration bpRegistration = bpos.createRegistration() //创建一个登记请求
>> //Use returned registration from namenode with updated fields
>> bpRegistration = bpNamenode.registerDatanode(bpRegistration)
                    //通过 proxy 远程调用 NameNode 上的 registerDatanode()
>> bpos.registrationSucceeded(this, bpRegistration)
>> scheduleBlockReport(dnConf.initialBlockReportDelay) //安排下次发送报告的时间
                    // random short delay - helps scatter the BR from all DNs

```

可见,DataNode 的初始化有相当大一块是在这里完成的,包括对 addBlockPool() 和 initStorage() 的调用,还有 initPeriodicScanners() 都是在这里完成的。虽然这都只是针对具体 nameservice 的,但实际上多数 Hadoop 集群中都只有一个 nameservice。不过请注意,如果同一个 nameservice 中有两个 NameNode(一主一备),那么只有第一个连上 NameNode 的 BPSERVICEActor 才会进行这些初始化,后面的就不会了。

由于前面已经与 NameNode 建立了 RPC 层的连接,这里的请求版本信息和请求登记都是通过 RPC 进行,bpNamenode 就是具体 NameNode 的 RPC 代理。以 registerDatanode() 为例,这会通过 ProtoBuf 和 RPC 连接导致 NameNode 上 DatanodeProtocol 界面上的同名函数被调用,实际上是 NameNodeRpcServer.registerDatanode(),注意这是在 NameNode 上:

```

[DatanodeProtocolClientSideTranslatorPB.registerDatanode()
=> NameNodeRpcServer.registerDatanode()]

NameNodeRpcServer.registerDatanode(DatanodeRegistration nodeReg)
> verifySoftwareVersion(nodeReg)
> namesystem.registerDatanode(nodeReg) //让 FSNamesystem 记下这个 DataNode
    == FSNamesystem.registerDatanode(DatanodeRegistration nodeReg)
>> getBlockManager().getDatanodeManager().registerDatanode(nodeReg)
    == DatanodeManager.registerDatanode(nodeReg) //DatanodeManager 是具体的主管部门
>> checkSafeMode()
> return nodeReg

```

NameNode 这边的程序从 NameNodeRpcServer.registerDatanode() 返回到 PB 层后,PB 层会把内容可能已经有所修改的 DatanodeRegistration 对象搭载在响应报文上发回给 DataNode,于是 DataNode 那一边的 BPSERVICEActor 线程就会被唤醒而从这 RPC 调用返回,

这就回到了前面 `connectToNNAndHandshake()` > `register()` 的代码中。本来,既然已经登了记,就可立即开始发送报告了,但是这里通过 `scheduleBlockReport()` 安排了一段随机长度的延迟,到那时才发送报告。之所以需要推迟一段随机长度的时间,是因为 DataNode 向 NameNode 的第一个报告通常会是比较大的,将会有较大的流量发送给 NameNode,如果集群中所有的 DataNode 都在差不多同一个时间开机,同一个时间向 NameNode 登记,那么 NameNode 就可能一下子应付不过来了。而引入一段随机长度的延迟,则有利于使后面较大的流量互相错开。

在 DataNode 这一边,与 NameNode 建立了连接并向其登记之后,相应的 BPSERVICEActor 就进入了它的主循环,反复地调用 `offerService()`,意为提供服务。不过一般提供服务的一方总是被动的,有要求来才服务,无要求则不服务;而 BPSERVICEActor 却是主动向 NameNode 发送心跳和提供报告,并执行 NameNode 发回的命令,实际上是主动要求提供服务。

我们看 `offerService()` 的代码摘要:

```
[BPSERVICEActor.run() > offerService()]

BPSERVICEActor.offerService()
> while (shouldRun()) {
>+ startTime = now()
>+ if (startTime - lastHeartbeat >= dnConf.heartBeatInterval) { //如果时间到点
>++ HeartbeatResponse resp = sendHeartBeat() //发送心跳报告,并接收回应
>++ bpos.updateActorStatesFromHeartbeat(this, resp.getNameNodeHaState())
>++ state = resp.getNameNodeHaState().getState()
>++ if (state == HASEVICEState.ACTIVE) {
>+++ handleRollingUpgradeStatus(resp)
>++ }
>++ processCommand(resp.getCommands()) //执行来自 NameNode 的命令
>++ if (sendImmediateIBR ||
    (startTime - lastDeletedReport > dnConf.deleteReportInterval)) {
>+++ reportReceivedDeletedBlocks()
>++ }
>++ List<DatanodeCommand> cmds = blockReport()
    //如果时间到点就向 NameNode 提交数据块存储报告并接收回应
>++ processCommand(cmds == null?
    null : cmds.toArray(new DatanodeCommand[cmds.size()]))
    //如果发送了报告,就执行 NameNode 搭载在回应中的命令
>++ DatanodeCommand cmd = cacheReport()
    //如果时间到点就向 NameNode 提交数据块缓存报告并接收回应
>++ processCommand(new DatanodeCommand[] { cmd })
    //如果发送了报告,就执行 NameNode 搭载在回应中的命令
>++ if (dn.blockScanner != null) {
```

```

>+++ dn.blockScanner.addBlockPool(bpos.getBlockPoolId())
//将扫描发现的块文件加到 BlockPool 中
>+ }
>+ pendingIncrementalBRperStorage.wait(waitTime) //睡眠一段时间
>+ } // end if (startTime - lastHeartbeat >= dnConf.heartBeatInterval)
> } // end while

```

在 offerService() 的循环中,表面上发送心跳的次数与发送 blockReport 和 cacheReport 的次数是一样的,然而实际上 blockReport() 和 cacheReport() 的内部还有附加条件,二者还有其自己的周期,所以实际上不一定像心跳那么频繁。当然,如果实际上没有发送,那么返回的 cmds 将是 null,而以 null 为参数调用 processCommand() 也不会有什么实际的操作。

13.2 心跳 HeartBeat

如上所述,DataNode 上对于每个 NameNode 都有个 BPServiceActor 线程,这个线程会定期向其 NameNode 发送心跳报告。现在我们就来看一下心跳报告的发送:

```
[BPServiceActor.run() > offerService() > sendHeartBeat()]
```

```

BPServiceActor.sendHeartBeat()
> StorageReport[] reports = dn.getFSDataset().getStorageReports(bpos.getBlockPoolId())
>> FsDatasetImpl.getStorageReports(String bpid)
>>> reports = new StorageReport[volumes.volumes.size()] //创建一个数组
>>> for (FsVolumeImpl volume : volumes.volumes) { //对于本节点上的每个文件卷
>>>+ reports[i++] = new StorageReport(volume.toDatanodeStorage(), false,
        volume.getCapacity(), volume.getDfsUsed(), volume.getAvailable(),
        volume.getBlockPoolUsed(bpid)) //从 FsVolumeImpl 中提取信息
>>> } //填写 StorageReport 数组 reports,这个数组是心跳报文的核心
> //通过 PB 层发送心跳报文,实质上是对 NameNode 的 RPC 调用
> bpNamenode.sendHeartbeat(bpRegistration, reports,
        dn.getFSDataset().getCacheCapacity(), dn.getFSDataset().getCacheUsed(), ...)
    == DatanodeProtocolClientSideTranslatorPB.sendHeartbeat(bpRegistration, reports, ...)
    // bpNamenode 是 NameNode 的 RPC 代理,这是个 ProtoBuf 层的模块
>> builder = HeartbeatRequestProto.newBuilder()
        .setRegistration(PBHelper.convert(registration))
        .setXmitsInProgress(xmitsInProgress)
        .setXceiverCount(xceiverCount)
        .setFailedVolumes(failedVolumes) //构建报文的头部
>> builder.addAllReports(PBHelper.convertStorageReports(reports)) //将 reports 加入报文
>> if (cacheCapacity != 0) builder.setCacheCapacity(cacheCapacity) //将缓存容量加入报文
>> if (cacheUsed != 0) builder.setCacheUsed(cacheUsed) //将已耗用缓存容量加入报文

```

```

>> resp = rpcProxy.sendHeartbeat(NULL_CONTROLLER, builder.build())
        //将 RPC 报文发往 NameNode,并等待对方发回响应报文 resp
>> DatanodeCommand[] cmds = new DatanodeCommand[resp.getCmdsList().size()]
        //从响应报文中获取命令数组大小,并按此大小创建数组 cmds
>> for (DatanodeCommandProto p : resp.getCmdsList()) {
        //从响应报文中抽取所搭载的命令数组
>>+ cmds[index] = PBHelper.convert(p) //并逐条进行格式转换
>>+ index++
>> }
>> if (resp.hasRollingUpgradeStatus()) {
>>+ rollingUpdateStatus = PBHelper.convert(resp.getRollingUpgradeStatus())
>> }
>> return new HeartbeatResponse(cmds,
        PBHelper.convert(resp.getHaStatus()), rollingUpdateStatus)
        //创建一个(DataNode 上的)HeartbeatResponse 对象,将其返回给 BPServiceActor

```

我们说 bpNamenode 是 NameNode 的 RPC 代理,可是这里又有个 rpcProxy,其实这并不冲突,只是层次不同而已。

从 DataNode 发往 NameNode 的心跳报文,其核心是一个 StorageReport 数组 reports。DataNode 上有多少个文件卷用于这个 NameNode 的 BlockPool,这个数组中就有多少个元素,即 StorageReport 对象。StorageReport 的数据结构部分是这样的:

```

class StorageReport{
] DatanodeStorage storage //一个 DatanodeStorage 对象
] String storageID //存储设备的 ID
] State state //状态,NORMAL、READ_ONLY_SHARED、FAILED
] StorageType storageType //存储类型,DISK、RAM_DISK、SSD 等
] boolean failed //设备是否失败
] long capacity //设备(文件卷)容量
] long dfsUsed //已由 HDFS 用去的容量
] long remaining //剩余容量
] long blockPoolUsed //由这个 BlockPool 用去的容量

```

代码中对 rpcProxy.sendHeartbeat()的调用是阻塞的,DataNode 这一边的 BPServiceActor 线程通过 PB 和 RPC 机制将心跳报文发送到 NameNode 这一边,然后就睡眠等待 NameNode 发回响应报文,收到了响应报文以后才从 rpcProxy.sendHeartbeat()返回,然后再继续往下执行,直至从 BPServiceActor.sendHeartBeat()返回。

按说既然 DataNode 这一边说是 sendHeartBeat(),那么在 NameNode 这一边就理应是 receiveHeartBeat(),但偏偏也是叫 sendHeartBeat()。RPC 要求两边的函数名相同,如此则形式上是在调用本地的函数,实际上却调到了对方的同名函数中。但是,即便如此,就称之为 HeartBeat()也不至于像 sendHeartBeat()那样带有误导性,使人一看到这个函数名还以为 NameNode 也会发送心跳信号。

NameNode 这一边由 NameNodeRpcServer 负责接受和处理来自各个 DataNode 的心跳和报告并做出反应,所以 NameNode 上的 PB 层接收到来自 DataNode 的心跳报文以后会调用 NameNodeRpcServer 的 sendHeartbeat() 方法,这里只是列出了 NameNode 上 PB 层的调用路径,PB 层以下那就是 IPC 和网络通信的事了:

```
[DatanodeProtocolServerSideTranslatorPB.sendHeartbeat()
=> NameNodeRpcServer.sendHeartbeat()]

NameNodeRpcServer.sendHeartbeat(DatanodeRegistration nodeReg,
                                StorageReport[] report, ...)

> verifyRequest(nodeReg)    //验证心跳来源的有效性
> namesystem.handleHeartbeat(nodeReg, report, ...)
== FSNamesystem.handleHeartbeat(...)

>> maxTransfer = blockManager.getMaxReplicationStreams() - xmitsInProgress
>> DatanodeCommand[] cmds = blockManager.getDatanodeManager().handleHeartbeat(...)
== DatanodeManager.handleHeartbeat(...)
    //由 DatanodeManager 加以处理,并返回一组对于 DataNode 的命令
>> NNHStatusHeartbeat haState = new NNHStatusHeartbeat(...) //与 HA 容错有关
>> return new HeartbeatResponse (cmds, haState, rollingUpgradeInfo)
    //创建一个响应报文,将命令搭载在上面,返回后由 PB 层发回 DataNode
```

调用参数其实有不少,这里没有详细列出,这些参数都来自 DataNode,是由 PB 层的 DatanodeProtocolServerSideTranslatorPB.sendHeartbeat() 从报文中解析出来的。

对于通过心跳发来的报告,NameNodeRpcServer 通过 FSNamesystem.handleHeartbeat() 加以处理,并因此而产生一组对于 DataNode 的命令 cmds,让对方执行。所生成的一组命令被搭载在 NameNode 的响应报文上,程序返回 PB 层后就发回给对方,即 DataNode 一方。

这里的核心的操作是由 DatanodeManager 完成的,DatanodeManager 是 NameNode 上专门管理 DataNode 的部件。对于 NameNode 这当然是个重要的部件,因为文件系统的数据都在 DataNode 上。下面是就我们此刻所关心问题的角度对 DatanodeManager 所作摘要:

```
class DatanodeManager {
] Namesystem namesystem //说到底,DatanodeManager 是为文件系统服务的
] BlockManager blockManager
    // DatanodeManager 是由 BlockManager 创建的,后者则由 FSNamesystem 创建
] HeartbeatManager heartbeatManager //由 DatanodeManager 创建
] NavigableMap<String, DatanodeDescriptor> datanodeMap
    //给定一个 StorageID,就可以查到其所在节点的 DatanodeDescriptor
] NetworkTopology networkTopology
] Host2NodesMap host2DatanodeMap //可以根据节点名查找其 DatanodeDescriptor
] DNSToSwitchMapping dnsToSwitchMapping
] HostFileManager hostFileManager
] HostSet includes //包括在内的所有 DataNode,类似于白名单
```

```
] HostSet excludes           //排除在外的所有 DataNode,类似于黑名单
```

```
] handleHeartbeat(DatanodeRegistration nodeReg, StorageReport[] reports, String blockPoolId,
    long cacheCapacity, long cacheUsed, int xceiverCount, int maxTransfers, int failedVolumes)
] ...
```

这样,只要有个 StorageID,实际上是 DataNode 的 Uuid,就可以在 DatanodeManager 的 datanodeMap 中找到该节点的 DatanodeDescriptor,进一步在这个 DatanodeDescriptor 的 storageMap 中可以找到这个节点上各个存储设备(文件卷)的 DatanodeStorageInfo。

```
class DatanodeDescriptor extends DatanodeInfo {    //代表着一个 DataNode,这是在 NameNode 上
] static class BlockTargetPair {    //块及其目标存放地点,由一对 block 和 targets 构成
] Block block                        //对于这个块的简单描述,包括
] long blockId
] long numBytes
] long generationStamp
] DatanodeStorageInfo[] targets    //复制这个块的若干目标地,每个目标包括下列信息
] DatanodeDescriptor dn
] String storageID
] StorageType storageType
] State state
] long capacity
] long remaining
] long bandwidth
] BlockQueue<BlockTargetPair> replicateBlocks    //队列中的每个元素都是个 BlockTargetPair
    //每个元素都是需要从这个节点复制到别的节点以增添复份的块及其目标节点
] BlockQueue<BlockInfoUnderConstruction> recoverBlocks    //需要恢复原状的块
] LightweightHashSet<Block> invalidateBlocks    //需要从这个 DN 上撤销复份的块
] CachedBlocksList cached            //缓存在该 DN 节点内存中的块的清单
] CachedBlocksList pendingCached    //需要让该 DN 节点加以缓存的块的清单
] CachedBlocksList pendingUncached    //需要让该 DN 节点解除缓存的块的清单
-----
] static class BlockQueue<E> {}
] Queue<E> blockq = new LinkedList<E>()
] Map<String, DatanodeStorageInfo> storageMap
    //给定一个存储设备的 ID,就可以在这个 Map 中找到其 DatanodeStorageInfo
] CachedBlocksList pendingCached    //The blocks which we want to cache on this DataNode.
] CachedBlocksList cached    //The blocks which we know are cached on this datanode.
] CachedBlocksList pendingUncached
    //The blocks which we want to uncache on this DataNode
```

这里三个队列需要加点说明。

一个是 `replicateBlocks`, 这是个 `BlockTargetPair` 的队列, 每个 `BlockTargetPair` 都表明一个需要通过跨节点复制为之增添复份的块, 及其目的地 (所以是 `Block` 和 `Target` 的 `Pair`), 而当前的这个 `DataNode` 节点, 就作为复制的源头。如果 `BlockManager` 发现某个块的复份数量不足, 就会为其找到一个合适的源头和目的地, 然后将一个 `BlockTargetPair` 挂入该源头节点的 `DatanodeDescriptor` 对象中的 `replicateBlocks` 队列。注意, `BlockTargetPair` 中提供的目的地 `Target` 可以不止一个, 因为有时候可能需要增添不止一个复份。后面我们还要在这个方向上深入到有关的细节中, 介绍 `BlockManager` 内部的 `ReplicationMonitor` 线程和 `computeReplicationWorkForBlocks()`、`BlockPlacementPolicyDefault.chooseTarget()` 等函数的代码, 现在暂且只要知道有这些考虑就行。

第二个是 `recoverBlocks`, 这是个需要恢复原状的块的队列。什么是需要恢复原状的块呢? 比方说有个 App 打开了一个文件, 在它的末尾即最后一块中添上了 2MB 的内容, 然后这个 App 崩溃了, 但文件没有关闭, 文件系统给这个 App 客户的“租约 (Lease)”就超时了。在这样的情况下, 那个 App 对此文件的操作理应作废, 被它改动过的那个块就得恢复原状。当然, 如果也有新增加的块, 那么新增的块也应该作废。

第三个是 `invalidateBlocks`, 这就是需要从这 `DataNode` 上撤销作废的块的队列。当然, 实际上只是这个块的一个复份。

注意, `BlockTargetPair` 是 `Block` 和 `Target` 的 `Pair`, 这里的 `Target` 是个 `DatanodeStorageInfo` 对象的数组。 `DatanodeStorageInfo` 中的信息既包括目标节点, 也包括对于具体存储设备 (即目标节点上的文件卷) 及其类型的说明。下面是 `DatanodeStorageInfo` 类的数据结构摘要:

```
class DatanodeStorageInfo { } //代表着 DataNode 上的一个存储设备
] DatanodeDescriptor dn
] String storageID
] StorageType storageType //例如 DISK, RAMDISK 等
] State state
] long capacity           //该设备的容量
] long dfsUsed            //已用去容量
] long remaining          //尚存容量
] blockPoolUsed
] BlockInfo blockList     //存储在这个设备上所有块的队列
```

这里的 `BlockInfo` 对象 `blockList` 含有的信息包括: 这个块属于哪一个 `INodeFile`, 以及它的复份都在哪些 `DataNode` 上。

```
class BlockInfo extends Block implements LightweightGSet.LinkedElement { } //数据结构
] BlockCollection bc // BlockCollection 是个 interface, INodeFile 实现了这个 interface
//把文件看成块的 (有序) 集合, 提供对于文件的有关块的操作
] LightweightGSet.LinkedElement nextLinkedElement
] Object[] triplets
```

特别要说一下数组 `triplets`, 这个数组的类型是 `Object`。在 Java 语言中, `Object` 是所有类的祖先和源头, 所有别的类都是直接或间接地从 `Object` 派生出来的, 都是对 `Object` 类的扩充。

这就是说, triplets 数组的元素可以是任何类型。这个数组有多大呢? 这里没有说, 所以是可变大小的, 不过实际使用中这个数组的大小一定是 3 的倍数, 所以名叫 triplets。当然, 这是由定义于 BlockInfo 的操作方法决定和保证的。进一步, 如果一个块有 N 个复份, 那么它的这个数组的大小就是 $3 \times N$ 。

这个数组是干什么用的呢? 假设这个块有 N 个复份, 那么它的第 i 个复份($i = 0, 1, \dots, N-1$) 在这个数组中接连占用 3 个元素: 其中 triplets[$3 * i$] 指向代表着该复份所在存储设备的 DatanodeStorageInfo 对象, triplets[$3 * i + 1$] 指向存储在同一设备上的前一个块的 BlockInfo, triplets[$3 * i + 2$] 则指向后一个块的 BlockInfo。之所以要这样, 按源码中所加的注释, 是为了节省空间开销: 按现在这样平均每个复份占 16 字节; 而若按正规的方法(采用 Java 类 LinkedList)加以实现则平均每个复份需占 42 字节。考虑到一个集群中可能总共存储着数百万、数千万个复份, 所节省的空间确实甚为可观。

再看看 BlockInfo 的操作方法部分, 搞懂这些方法可以帮我们理解 triplets 这个方案是如何工作的, 更利于对后面那些代码的理解。

```
class BlockInfo extends Block implements LightweightGSet.LinkedElement {} //操作方法
] BlockInfo(Block blk, int replication) { //构造函数
    > super(blk) // BlockInfo 是对 Block 的扩充
    > this.triplets = new Object[3 * replication] // triplets 数组的大小是复份的个数乘以 3
    > this.bc = null
] getStorageInfo(int index)
    > return (DatanodeStorageInfo)triplets[index * 3] //这个元素指向一个 DatanodeStorageInfo
] getDatanode(int index)
    > DatanodeStorageInfo storage = getStorageInfo(index)
    > return storage == null ? null : storage.getDatanodeDescriptor()
//进一步可获取所在节点的 DatanodeDescriptor
] getPrevious(int index)
    > BlockInfo info = (BlockInfo)triplets[index * 3 + 1] //这一元素指向其前一块的 BlockInfo
    > return info
] getNext(int index)
    > BlockInfo info = (BlockInfo)triplets[index * 3 + 2] //这一元素指向其后一块的 BlockInfo
    > return info
] findStorageInfo(DatanodeStorageInfo storageInfo)
    //从 BlockInfo 中找到其存储在给定设备上的复份, 返回其数组下标
    > len = getCapacity()
    > for(int idx = 0; idx < len; idx++) {
    >+ DatanodeStorageInfo cur = getStorageInfo(idx)
    >+ if(cur == storageInfo) return idx
    >+ if(cur == null) break
    > }
    > return -1
```

```

] listInsert(BlockInfo head, DatanodeStorageInfo storage)
    //Insert this block into the head of the list of blocks related to the specified
    //DatanodeStorageInfo. If the head is null then form a new list.
    //Return current block as the new head of the list.
    > dnIndex = this.findStorageInfo(storage)
    > this.setPrevious(dnIndex, null)
    > this.setNext(dnIndex, head)
    > if(head!= null) head.setPrevious(head.findStorageInfo(storage), this)
    > return this
] removeStorage(DatanodeStorageInfo storage)
    > int dnIndex = findStorageInfo(storage)
    > if(dnIndex < 0) return false //the node is not found
    > lastNode = numNodes() - 1
    > // replace current node triplet by the lastNode one
    > setStorageInfo(dnIndex, getStorageInfo(lastNode))
    > setNext(dnIndex, getNext(lastNode))
    > setPrevious(dnIndex, getPrevious(lastNode))
    > // set the last triplet to null
    > setStorageInfo(lastNode, null)
    > setNext(lastNode, null)
    > setPrevious(lastNode, null)
    > return true
] moveBlockToHead(BlockInfo head, DatanodeStorageInfo storage, int curIndex, int headIndex)
    > if (head == this) return this
    > BlockInfo next = this.setNext(curIndex, head)
    > BlockInfo prev = this.setPrevious(curIndex, null)
    > head.setPrevious(headIndex, this)
    > prev.setNext(prev.findStorageInfo(storage), next)
    > if (next!= null) next.setPrevious(next.findStorageInfo(storage), prev)
    > return this //返回这个 BlockInfo 本身,成为所在 storage 这个队列的头

```

现在我们回到 DatanodeManager 里的 HeartbeatManager,这是由 DatanodeManager 创建的,专门用来处理来自 DataNode 的心跳。

```

class HeartbeatManager implements DatanodeStatistics {}
] List<DatanodeDescriptor> datanodes //作用跟 datanodeMap 相似,但这是有先后次序的
] Daemon heartbeatThread = new Daemon(new Monitor()) //心跳监视线程,类型定义见下
] Namesystem namesystem //指向所在 NameNode 上的文件系统
] BlockManager blockManager
-----
] addDatanode(final DatanodeDescriptor d) //将一 DataNode 加入 datanodes 这个 List 中

```

```

] heartbeatCheck()           //Check if there are any expired heartbeats,
                             //and if so, whether any blocks have to be re-replicated.
] updateHeartbeat(nodeinfo, reports, cacheCapacity, cacheUsed, ...)
] class Monitor implements Runnable {} //对于心跳监视线程这个类的定义
]] run()
  > while(namesystem.isRunning()) {
  >+ if (lastHeartbeatCheck + heartbeatRecheckInterval < now) {
  >++ heartbeatCheck()
  >++ lastHeartbeatCheck = now
  >+ }
  >+ if (blockManager.shouldUpdateBlockKey(now - lastBlockKeyUpdate)) {
  >++ for(DatanodeDescriptor d : datanodes) d.needKeyUpdate = true
  >+ }
  >+ Thread.sleep(5000); // 5 seconds
  > }

```

其实 NameNode 上的 HeartbeatManager 应该叫 HeartbeatMonitor 或 HeartbeatChecker, 这是一个用来监视和检查心跳的线程。从这个线程的 run() 函数的代码中可以看出, 其主循环就是定期调用 heartbeatCheck()。检查什么呢? 检查该来的心跳是否如期到来。当然, 心跳的周期未必很精确, 但若是一个 DataNode 的心跳缺失了, 超时太久仍未到来, 就说明这个节点可能出问题了。这就麻烦了, 因为这个节点上可能存储着成千上万个数据块的复份, 这一来那些复份就丢失了。幸好每个块都是“狡兔三窟”, 不止一个复份, 但是这样一来那些块的复份就都少了一个, 需要补上。所以就要为这些数据块从别的复份再拷贝一份到别的节点上, 使复份的个数恢复到预定的数量。可是, 说不定过一会儿那个节点又回来了, 于是这些块的复份又太多了, 得要撤销(invalidate)一个才对, 否则就太占存储空间了。所以这里可能会有不少的活动, 这些相关的处理和活动总的来说都是靠 BlockManager 完成的, 我们先看一下 BlockManager 数据结构部分的摘要:

```

class BlockManager{} //数据结构部分
] Namesystem namesystem //指向所在 NameNode 上的文件系统
] DatanodeManager datanodeManager
] HeartbeatManager heartbeatManager
] BlocksMap blocksMap //给定一个 Block 对象, 就可找到其相应的 BlockInfo 对象
] CorruptReplicasMap corruptReplicas //已损坏复份的集合
//给定一个 Block, 可以找到其损坏的复份在哪些节点上, 以及损坏原因
] InvalidateBlocks invalidateBlocks //需要从 DataNode 上注销的复份
//给定一个 DatanodeInfo, 可以查到在该节点上的哪些块(复份)需被注销
] Set<Block> postponedMisreplicatedBlocks //用于 NameNode 倒换时的处理
] Map<String, LightweightLinkedSet<Block>> excessReplicateMap
//复份已经过量的块的 Map
] UnderReplicatedBlocks neededReplications //复份数量不足的块的集合, 分优先级排列

```



```
] PendingReplicationBlocks pendingReplications //已通知复制但尚未报告完成的块的集合
] Daemon replicationThread = new Daemon(new ReplicationMonitor()) //备份监督线程
```

摘要中已加足够注释,无须再多说了。

现在,我们可以回过去,继续往下看对于 `DatanodeManager.handleHeartbeat()` 的调用了:

```
[DatanodeProtocolServerSideTranslatorPB.sendHeartbeat()
=> NameNodeRpcServer.sendHeartbeat() > FSNamesystem.handleHeartbeat()
> DatanodeManager.handleHeartbeat()]

DatanodeManager.handleHeartbeat (DatanodeRegistration nodeReg, StorageReport[] reports,
    long cacheCapacity, long cacheUsed, int xceiverCount, int maxTransfers, int failedVolumes)
> nodeinfo = getDatanode(nodeReg)
    //nodeinfo 来自 DataNode 的登记信息,来自 DataNode 的报告都带有这个信息
> if (nodeinfo != null && nodeinfo.isDisallowed()) { //节点已登记,但已禁用
>+ setDatanodeDead(nodeinfo)
>+ DisallowedDatanodeException(nodeinfo) //那就异常返回
> }
> if (nodeinfo == null || !nodeinfo.isAlive) //如果 DataNode 尚未登记
>+ return new DatanodeCommand[]{RegisterCommand.REGISTER} //就让它先登记再报告
> //DataNode 已经登记,根据心跳报告更新有关具体 DataNode 的种种信息和统计
> heartbeatManager.updateHeartbeat(nodeinfo, reports, cacheCapacity, cacheUsed, ...)
    //根据心跳报告的内容更新关于该 DataNode 的记载
>> stats.subtract(node) //先从统计信息中减去该减的
>> node.updateHeartbeat(reports, cacheCapacity, cacheUsed, xceiverCount, failedVolumes)
    == DatanodeDescriptor.updateHeartbeat(reports, cacheCapacity, cacheUsed, ...)
>>> updateHeartbeatState(reports, cacheCapacity, cacheUsed, xceiverCount, volFailures)
>>>> checkFailedStorages =
    (volFailures > this.volumeFailures) || !heartbeatedSinceRegistration
>>>> if (checkFailedStorages) {
>>>>+ LOG.info("Number of failed storage changes from " + this.volumeFailures
    + " to " + volFailures)
>>>>+ failedStorageInfos = new HashSet<DatanodeStorageInfo>(storageMap.values())
>>>> }
>>>> setCacheCapacity(cacheCapacity) //根据 DataNode 的报告设置其缓存容量
>>>> setCacheUsed(cacheUsed)
>>>> setXceiverCount(xceiverCount)
>>>> setLastUpdate(Time.now())
>>>> this.volumeFailures = volFailures
>>>> for (StorageReport report : reports) { //对于心跳报告中关于每一个存储设备的报告
>>>>+ DatanodeStorageInfo storage = updateStorage(report.getStorage())
```

//更新有关该设备的信息

```

>>>>+> DatanodeStorageInfo storage = storageMap.get(s.getStorageID())
>>>>+> if (storage == null) {
>>>>+>+ LOG.info("Adding new storage ID " + s.getStorageID()
                                + " for DN " + getXferAddr())
>>>>+>+ storage = new DatanodeStorageInfo(this, s)
>>>>+>+ storageMap.put(s.getStorageID(), storage)
>>>>+> } else if (storage.getState() != s.getState() ||
                                storage.getStorageType() != s.getStorageType()) {
>>>>+>+ storage.updateFromStorage(s)
>>>>+>+ storageMap.put(storage.getStorageID(), storage)
>>>>+> }
>>>>+> return storage
>>>>+ if (checkFailedStorages) failedStorageInfos.remove(storage)
>>>>+ storage.receivedHeartbeat(report) == DatanodeStorageInfo.receivedHeartbeat(report)
>>>>+> updateState(report) //更新该存储设备的容量信息
>>>>+> heartbeatedSinceFailover = true
>>>>+ totalCapacity += report.getCapacity()
>>>>+ totalRemaining += report.getRemaining()
>>>>+ totalBlockPoolUsed += report.getBlockPoolUsed()
>>>>+ totalDfsUsed += report.getDfsUsed()
>>>> }
>>>> rollBlocksScheduled(getLastUpdate())
>>>> setCapacity(totalCapacity)
>>>> setRemaining(totalRemaining)
>>>> setBlockPoolUsed(totalBlockPoolUsed)
>>>> setDfsUsed(totalDfsUsed)
>>>> if (checkFailedStorages) updateFailedStorage(failedStorageInfos)
>>>> heartbeatedSinceRegistration = true
>> stats.add(node) //与前面的 stats.subtract(node)相对应
> if(namesystem.isInSafeMode()) return new DatanodeCommand[0]
    //是 SafeMode 就返回一个空 DatanodeCommand 数组,对 DataNode 没有命令
> //不是 SafeMode,准备要发送给 DataNode 的命令
> BlockInfoUnderConstruction[] blocks =
    nodeinfo.getLeaseRecoveryCommand(Integer.MAX_VALUE)
>> blocks = recoverBlocks.poll(maxTransfers) //获取需要恢复原状的数据块列表
>> return blocks.toArray(new BlockInfoUnderConstruction[blocks.size()])
> if (blocks != null) { //check lease recovery,有数据块需要恢复原状,加上恢复命令
>+ brCommand = new BlockRecoveryCommand(blocks.length)
>+> super(DatanodeProtocol.DNA_RECOVERBLOCK)

```

```

>+ for (BlockInfoUnderConstruction b : blocks) {
>+ ...
>+ brCommand.add (new RecoveringBlock(new ExtendedBlock(blockPoolId, b),
    DatanodeStorageInfo.toDatanodeInfos(recoveryLocations),
    b.getBlockRecoveryId()))
>+ }
> }
> cmds = new ArrayList<DatanodeCommand>() //创建一个命令块
> List<BlockTargetPair> pendingList = nodeinfo.getReplicationCommand(maxTransfers)
    //获取需要补充复份的数据块列表,注意这是一个 BlockTargetPair 的列表
>> return replicateBlocks.poll(maxTransfers) //replicateBlocks 队列就是需要添加的队列
> if (pendingList!= null) { //有数据块需要补充复份,加上传输复制的命令
>+ cmds.add(new BlockCommand(
    DatanodeProtocol.DNA_TRANSFER, blockPoolId, pendingList))
> }
> Block[] blks = nodeinfo.getInvalidateBlocks(blockInvalidateLimit)
    //check block invalidation,获取需要予以撤销的复份,这是个 Block 对象数组
>> Block[] deleteList = invalidateBlocks.pollToArray(new Block[Math.min(
    invalidateBlocks.size(), maxblocks)])
>> return deleteList.length == 0?null : deleteList
> if (blks!= null) { //有需要撤销的数据块,加上撤销数据块复份的命令
>+ cmds.add(new BlockCommand(DatanodeProtocol.DNA_INVALIDATE, blockPoolId, blks))
> }
> if (shouldSendCachingCommands && ...){ //有需要(在内存中)加以缓存的复份
>+ DatanodeCommand pendingCacheCommand =
    getCacheCommand(nodeinfo.getPendingCached(), nodeinfo,
    DatanodeProtocol.DNA_CACHE, blockPoolId) //需要加以缓存的数据块
>+ if (pendingCacheCommand!= null) {
>+ cmds.add(pendingCacheCommand) //如果有加缓存要求就加到命令块中
>+ sendingCachingCommands = true
>+ }
>+ pendingUncacheCommand = getCacheCommand(nodeinfo.getPendingUncached(),
    nodeinfo, DatanodeProtocol.DNA_UNCACHE, blockPoolId) //需要解除缓存的数据块
>+ if (pendingUncacheCommand!= null) {
>+ cmds.add(pendingUncacheCommand) //如果有去缓存要求就加到命令块中
>+ sendingCachingCommands = true
>+ }
>+ if (sendingCachingCommands) nodeinfo.setLastCachingDirectiveSentTimeMs(nowMs)
> }
> blockManager.addKeyUpdateCommand(cmds, nodeinfo) //如果有密钥的变动

```

```

> if (nodeinfo.getBalancerBandwidth() > 0) { //如果需要改变网络通信的带宽
>+ cmds.add(new BalancerBandwidthCommand(nodeinfo.getBalancerBandwidth()))
>+ nodeinfo.setBalancerBandwidth(0)
> }
> if (!cmds.isEmpty()) return cmds.toArray(new DatanodeCommand[cmds.size()])
//cmds 非空,将其转换成一个 DatanodeCommand 数组并返回
> return new DatanodeCommand[0] //cmds 空,就返回一个空 DatanodeCommand 数组

```

这个函数的调用参数有不少,都是从心跳报文中解析出来的,源头都在 DataNode 那一边,这里不再详细列举和解释,有需要的读者可以直接参阅源码。

NameNode 对于心跳的处理主要有两个方面:一是根据收到的报告更新具体 DataNode 留在 NameNode 上的种种信息,使其更符合此刻的实际情况;二是根据两边的情况生成一组行动命令,搭载在响应报文上发回,让 DataNode 照办。相比之下,前者是直截了当的事,没有什么可说的;而后者就比较复杂并且重要了。

节点之间的通信当然得有协议(规程)即 Protocol,而协议又是有层次的,我们在这里关心的当然不是 TCP 这个层次上的,而是应用层的协议。就心跳这个事情而言,DataNode 向 NameNode 发送的是 HeartbeatRequestProto 报文,反过来 NameNode 发回的响应是 HeartbeatResponseProto 报文。而 NameNode 对于 DataNode 的命令,则遵循 DatanodeProtocol 协议搭载在响应报文上,该协议定义了一些行动命令:

```

interface DatanodeProtocol {}
] DNA_UNKNOWN = 0; // unknown action
] DNA_TRANSFER = 1; // transfer blocks to another datanode
] DNA_INVALIDATE = 2; // invalidate blocks
] DNA_SHUTDOWN = 3; // shutdown node
] DNA_REGISTER = 4; // re-register
] DNA_FINALIZE = 5; // finalize previous upgrade
] DNA_RECOVERBLOCK = 6; // request a block recovery
] DNA_ACCESSKEYUPDATE = 7; // update access key
] DNA_BALANCERBANDWIDTHUPDATE = 8; // update balancer bandwidth
] DNA_CACHE = 9; // cache blocks
] DNA_UNCACHE = 10; // uncache blocks

```

DNA 应该是“DataNode Action”的缩写,就是要求 DataNode 采取的行动。不过这只是命令的一部分,相当于一个操作码,而整个命令则是一个 ServerCommand 对象(可以理解成数据结构)。

以 BlockCommand 为例,其 Action 代码可以是 DNA_TRANSFER,也可以是 DNA_RECOVERBLOCK 或 DNA_INVALIDATE 等,但是所携带的信息当然不只是一个 Action 代码,还得要有关于块的描述。BlockCommand 的定义是这样来的:

```

abstract class ServerCommand {}
] int action //抽象类 ServerCommand 只含有一个 Action 代码

```

```

] ServerCommand(int action)    //构造方法
    > this.action = action

abstract class DatanodeCommand extends ServerCommand {}
    //抽象类 DatanodeCommand 是对 ServerCommand 的扩充,但是并未增添任何信息
] DatanodeCommand(int action)
    > super(action)

public class BlockCommand extends DatanodeCommand {}
    // BlockCommand 是对 DatanodeCommand 的扩充,增添了数据块转移所需的信息
] String poolId                //BlockPool 的 Id
] Block[] blocks               //一组数据块描述
] DatanodeInfo[][] targets     //各数据块复份的目标节点,也许需要增添多个复份
] StorageType[][] targetStorageTypes //各数据块复份的存储类型
] String[][] targetStorageIDs   //各数据块复份的存储设备 Id
] BlockCommand(int action, String poolId, List<BlockTargetPair> blocktargetlist)
    //构造方法,用于 DNA_TRANSFER
    > super(action)
    > this.poolId = poolId
    > blocks = new Block[blocktargetlist.size()]           //这是个一维数组
    > targets = new DatanodeInfo[blocks.length][]         //这是个二维数组
    > targetStorageTypes = new StorageType[blocks.length][] //这也是个二维数组
    > targetStorageIDs = new String[blocks.length][]       //这也是个二维数组
    > for(int i = 0; i < blocks.length; i++) {
    >+ BlockTargetPair p = blocktargetlist.get(i)
    >+ blocks[i] = p.block
    >+ targets[i] = DatanodeStorageInfo.toDatanodeInfos(p.targets)
    >+ targetStorageTypes[i] = DatanodeStorageInfo.toStorageTypes(p.targets)
    >+ targetStorageIDs[i] = DatanodeStorageInfo.toStorageIDs(p.targets)
    > }
] BlockCommand(int action, String poolId, Block blocks[])
    //另一个构造方法,用于 DNA_INVALIDATE
    > this(action, poolId, blocks, EMPTY_TARGET_DATANODES,
        EMPTY_TARGET_STORAGE_TYPES, EMPTY_TARGET_STORAGEIDS)
] BlockCommand(int action, String poolId, Block[] blocks, DatanodeInfo[][] targets,
    StorageType[][] targetStorageTypes, String[][] targetStorageIDs)
    > super(action)
    > this.poolId = poolId
    > this.blocks = blocks
    > this.targets = targets

```

```
> this.targetStorageTypes = targetStorageTypes
> this.targetStorageIDs = targetStorageIDs
```

注意 BlockCommand 中的 blocks 是个 Block 对象的一维数组,指明了本命令所涉及的所有数据块;而 targets 却是 DatanodeInfo 对象的二维数组,这是因为也许需要为一数据块增添不止一个的复份,这就需要有不止一个的目标节点。同样, targetStorageTypes 和 targetStorageIDs 也是二维数组,因为需要为每个复份指明其存储的类型和所在的(目标节点上的)设备。至于需要增添的每个数据块复份应该以何种存储类型放在哪个节点上的什么设备上,我们在前面 DatanodeManager.handleHeartbeat() 的代码摘要中看到,这是由 DatanodeDescriptor.getReplicationCommand() 返回的,实际来自 DatanodeDescriptor 中的前述三个队列之一,这里就不深入进去了。

DNA_TRANSFER 命令是这样,别的命令也是大同小异,就留给读者举一反三了。

进一步,NameNode 对于心跳的反应是这样,可想而知对于 blockReport 和 cacheReport 的反应也是大同小异,无非就是更新关于具体 DataNode 的某些方面的信息,然后要是需要由 DataNode 采取什么行动就在响应报文中捎回相应的命令。即使没有命令,响应报文总归也是要发送的。

至于这些命令搭载在响应报文上回到 DataNode 上以后所引起的操作,这里就不深入下去了。

13.3 BlockReport

块报告 BlockReport 是对心跳报告的细化和补充,心跳所报告的只是对各存储设备状况的宏观统计,而 blockReport 所报告的却是具体每个块的存在和状态。注意,块报告的内容不仅仅来自块扫描,也来自目录扫描,是二者的综合。我们从具体 BPServiceActor 线程调用 blockReport() 开始:

```
[BPServiceActor.run() > offerService() > blockReport()]
```

```
BPServiceActor.blockReport()
```

```
> reportReceivedDeletedBlocks()
```

```
>> reports =
```

```
    new ArrayList<StorageReceivedDeletedBlocks>(pendingIncrementalBRperStorage.size())
```

```
    //内容来自 BPServiceActor 的 pendingIncrementalBRperStorage MAP
```

```
>> for (Map.Entry<DatanodeStorage, PerStoragePendingIncrementalBR> entry :
```

```
        pendingIncrementalBRperStorage.entrySet()) {
```

```
    //对于 pendingIncrementalBRperStorage MAP 中的每一项
```

```
>>+ DatanodeStorage storage = entry.getKey()
```

```
>>+ PerStoragePendingIncrementalBR perStorageMap = entry.getValue()
```

```
>>+ if (perStorageMap.getBlockInfoCount() > 0) { //如果该存储设备中有新的接收或删除
```

```
    //Send newly-received and deleted blockids to namenode
```

```
>>++ ReceivedDeletedBlockInfo[] rdbi = perStorageMap.dequeueBlockInfos()
```



```

>> ++ reports.add(new StorageReceivedDeletedBlocks(storage, rdbi)) //把它加到报告中
>> + } //end if
>> + sendImmediateIBR = false
>> } //end for
>> if (reports.size() == 0) return //如果没有新收到或删除的块,就不用发送
>> bpNamenode.blockReceivedAndDeleted(bpRegistration, bpos.getBlockPoolId(),
    reports.toArray(new StorageReceivedDeletedBlocks[reports.size()]))
    //通过 NameNode 的代理发送报告, RPC 调用 NameNode 上的这个函数
> Map<DatanodeStorage, BlockListAsLongs> perVolumeBlockLists =
    dn.getFSDataset().getBlockReports(bpos.getBlockPoolId())
    == FsDatasetImpl.getBlockReports(String bpid) //获取块报告内容
> // Convert the reports to the format expected by the NN.
> StorageBlockReport reports[] = new StorageBlockReport[perVolumeBlockLists.size()]
> for(Map.Entry<DatanodeStorage, BlockListAsLongs> kvPair :
    perVolumeBlockLists.entrySet()) {
> + BlockListAsLongs blockList = kvPair.getValue()
> + reports[i++] = new StorageBlockReport(kvPair.getKey(), blockList.getBlockListAsLongs())
> + totalBlockCount += blockList.getNumberOfBlocks()
> } //end for
> // Send the reports to the NN.
> if (totalBlockCount < dnConf.blockReportSplitThreshold) { //如果可以容纳于一个报文中
    // Below split threshold, send all reports in a single message.
> + cmd = bpNamenode.blockReport(bpRegistration, bpos.getBlockPoolId(), reports)
    //通过 NameNode 的 proxy 发送 RPC 请求,调用 NameNode 上的 blockReport()
> + if (cmd != null) cmds.add(cmd)
> } else { // Send one block report per message. 如果不能容纳,就要分批发送
> + for (StorageBlockReport report : reports) {
> ++ StorageBlockReport singleReport[] = { report }
> ++ cmd = bpNamenode.blockReport(bpRegistration, bpos.getBlockPoolId(), singleReport)
    //通过 NameNode 的 proxy 发送 RPC 请求,调用 NameNode 上的 blockReport()
> ++ if (cmd != null) cmds.add(cmd)
> + }
> }
> scheduleNextBlockReport(startTime) //安排下次提供块报告的时间
> return cmds.size() == 0 ? null : cmds

```

所谓向 NameNode 发送块报告,其实是两次报告,两次对 NameNode 的 RPC 调用。第一次是 reportReceivedDeletedBlocks(),报告自从上次报告以后新接收存储的块和按 NameNode 通知删除的块。其内容来自 BPServiceActor 对相关活动的记录。第二次才是普通意义上的块报告 blockReport,内容来源于 FsDatasetImpl.volumes。FsDatasetImpl 中的 volumes 是个 FsVolumeList,其中按本节点上不同的文件卷记载着每个数据块复份的 ReplicaInfo。这些复

份可能处于不同的状态,有的已经 Finalized,其块文件已经进入 Finalized 子目录;有的还在未定即“Under Construction”的状态。

```
[BPSERVICEACTOR.offerService() > blockReport() > FsDatasetImpl.getBlockReports()]

FsDatasetImpl.getBlockReports(String bpid)
> blockReportsMap = new HashMap<DatanodeStorage, BlockListAsLongs>()
> finalized = new HashMap<String, ArrayList<ReplicaInfo>>()
> uc = new HashMap<String, ArrayList<ReplicaInfo>>()
           //创建 finalized 和 uc 两个 Map,UC 即“Under Construction”。
> for (FsVolumeSpi v : volumes.volumes) { //先在这二者中为所有文件卷都创建一个 List
>+ finalized.put(v.getStorageID(), new ArrayList<ReplicaInfo>())
>+ uc.put(v.getStorageID(), new ArrayList<ReplicaInfo>())
> } //于是就有了已经 finalized 和尚在 under construction 的两套班子,但此刻都还是空的
> for (ReplicaInfo b : volumeMap.replicas(bpid)) {
           //扫描 volumeMap 中所有关于本 BlockPool 的复份的记载,
           //按该复份的状态分别加入 finalized 或 uc 中相应存储设备(即文件卷)的 List
>+ switch(b.getState()) {
>+ case FINALIZED: finalized.get(b.getVolume().getStorageID()).add(b)
>+ case RBW: case RWR: uc.get(b.getVolume().getStorageID()).add(b)
>+ case RUR:
>++ ReplicaUnderRecovery rur = (ReplicaUnderRecovery)b
>++ uc.get(rur.getVolume().getStorageID()).add(rur.getOriginalReplica())
>+ case TEMPORARY: break
>+ }
> } //end for
> for (FsVolumeSpi v : volumes.volumes) { //对于每个文件卷
>+ ArrayList<ReplicaInfo> finalizedList = finalized.get(v.getStorageID()) //获取其 finalizedList
>+ ArrayList<ReplicaInfo> ucList = uc.get(v.getStorageID()) //并获取其 ucList
>+ blockReportsMap.put(((FsVolumeImpl) v).toDatanodeStorage(),
           new BlockListAsLongs(finalizedList, ucList)) //并入 blockReportsMap
> }
> return blockReportsMap
```

这样,getBlockReports()就返回本节点上一个 BlockPool(实际上是 BlockPoolSlice)的完整的复份清单。这个清单是个 BlockListAsLongs,里面关于每个 Block 的信息都已转换成整数数组的形式,而不再是数据结构的形式,这样比较适合跨节点的发送。回到前面 blockReport()的摘要中,下面通过 RPC 调用提交报告,读者看一下所加的注释就可明白了。

到了 NameNode 这一边,跟 sendHeartbeat()的情况相似,也是由 NameNodeRpcServer 负责接受和处理来自各个 DataNode 的 blockReport()并做出反应。这是个跨节点调用的过程:

```
[DatanodeProtocolServerSideTranslatorPB.blockReport()
=> NameNodeRpcServer.blockReport()]

NameNodeRpcServer.blockReport(DatanodeRegistration nodeReg,
                                String poolId, StorageBlockReport[] reports)
> verifyRequest(nodeReg) //验证对方是个已经登记的 DataNode,如果不是就发起异常
> BlockManager bm = namesystem.getBlockManager() //NameNode 节点的 BlockManager
> boolean noStaleStorages = false
> for(StorageBlockReport r : reports) { //对于 reports 中关于每个存储设备的 List
>+ BlockListAsLongs blocks = new BlockListAsLongs(r.getBlocks())
    //将一个 StorageBlockReport 中的 BlockId 数组用作整数数组
    // BlockManager.processReport accumulates information of prior calls
    // for the same node and storage, so the value returned by the last
    // call of this loop is the final updated value for noStaleStorage.
>+ noStaleStorages = bm.processReport(nodeReg, r.getStorage(), blocks)
    == BlockManager.processReport(nodeReg, r.getStorage(), blocks)
> } //end for
> if (nn.getFSImage().isUpgradeFinalized() &&!namesystem.isRollingUpgrade()
    &&!nn.isStandbyState() && noStaleStorages) {
>+ return new FinalizeCommand(poolId) //若条件符合就创建 FinalizeCommand 作为回应
>+> super(DatanodeProtocol.DNA_FINALIZE) //具体命令是 DNA_FINALIZE
>+> blockPoolId = bpid //针对本 NameNode 的 blockPool
> }
> return null
```

NameNode 上出面接受和处理心跳的是 HeartbeatManager, 对于 blockReport 则是 BlockManager, 这里调用的是 BlockManager.processReport()。另外, 如果需要做出响应, 则这一次返回的命令是 DNA_FINALIZE; 而 BlockManager.processReport() 的返回值也是发出 DNA_FINALIZE 命令的条件之一。注意, 对 BlockManager.processReport() 的调用是在一个 for 循环中, 整个循环所针对的是同一个 DataNode, 但是每次都针对该节点上的一个不同的 DatanodeStorage, 例如具体的磁盘、SSD、RamDisk。代码中把每次调用这个函数的返回值赋给变量 noStaleStorages。于是问题就来了, 在这一轮循环中把函数的返回值赋给 noStaleStorages, 不是就把上一轮循环中的赋值给覆盖了? 所以代码中特别加了注释, 说 BlockManager.processReport() 内部会累积每次调用的结果, 因而以最后一次调用的返回值为准确就可以了。最后的这个返回值, 即 noStaleStorages, 是个布尔量, 这是 NameNode 向 DataNode 发出 DNA_FINALIZE 命令的一个必要条件。实际上其余那几个条件一般都是满足的, 因为 NameNode 一般都不是正在升级的过程中, 我们所考虑的也不是作为热备份而不应介入发号施令的那个 Standby NameNode (StandbyNN), 所以 noStaleStorages 实际上起着决定作用。那么所谓 noStaleStorages 是什么意思呢? 就是没有处于停运状态的存储设备, 如果 NameNode 发现来自 DataNode 的报告中缺失了关于某个(原来存在的)存储设备的内容,

就认为这个存储设备处于停运的状态。既然有存储设备处于停运状态,自然就不能命令它 DNA_FINALIZE。

我们看这里调用的 BlockManager.processReport(),这是带三个参数的 processReport():

```
[NameNodeRpcServer.blockReport() > BlockManager.processReport()]
```

```
BlockManager.processReport(DatanodeID nodeID,
```

```
    DatanodeStorage storage, BlockListAsLongs newReport)
```

```
    //注意这个 processReport() 有三个参数
```

```
> node = datanodeManager.getDatanode(nodeID) //获取对方的 DatanodeDescriptor
```

```
> if (node == null || !node.isAlive) {
```

```
    //如果 DatanodeManager 中没有对方的 DatanodeDescriptor,或尚未登记
```

```
>+ IOException("ProcessReport from dead or unregistered node: " + nodeID) //发起异常
```

```
> }
```

```
> DatanodeStorageInfo storageInfo = node.getStorageInfo(storage.getStorageID())
```

```
> if (storageInfo == null) storageInfo = node.updateStorage(storage)
```

```
> if (storageInfo.numBlocks() == 0) {
```

```
>+ processFirstBlockReport(storageInfo, newReport)
```

```
    //对于存储设备上的第一个 BlockReport 的处理可以简化,效率较高
```

```
> } else {
```

```
>+ processReport(storageInfo, newReport) //一般情况下的 BlockReport 处理
```

```
    //注意这个 processReport() 只有两个参数
```

```
>+> Collection<BlockInfo> toAdd = new LinkedList<BlockInfo>()
```

```
>+> Collection<Block> toRemove = new TreeSet<Block>()
```

```
>+> Collection<Block> toInvalidate = new LinkedList<Block>()
```

```
>+> Collection<BlockToMarkCorrupt> toCorrupt = new LinkedList<BlockToMarkCorrupt>()
```

```
>+> Collection<StatefulBlockInfo> toUC = new LinkedList<StatefulBlockInfo>()
```

```
>+> reportDiff(storageInfo, report, toAdd, toRemove, toInvalidate, toCorrupt, toUC)
```

```
    //对具体存储设备报告中的每一个块,扫描该设备的 storageInfo 队列
```

```
    //看对于这个块是否需要执行添加(toAdd)、删除(toRemove)等操作
```

```
>+> DatanodeDescriptor node = storageInfo.getDatanodeDescriptor()
```

```
>+> // Process the blocks on each queue
```

```
>+> for (StatefulBlockInfo b : toUC) { //对于 toUC 集合中的每一个块
```

```
>+>+ addStoredBlockUnderConstruction(b, storageInfo)
```

```
>+> }
```

```
>+> for (Block b : toRemove) { //对于 toRemove 集合中的每一个块
```

```
>+>+ removeStoredBlock(b, node)
```

```
>+> }
```

```
>+> for (BlockInfo b : toAdd) { //对于 toAdd 集合中的每一个块
```

```
>+>+ addStoredBlock(b, storageInfo, null, numBlocksLogged < maxNumBlocksToLog)
```

```
>+>+ numBlocksLogged ++
```

```

>+> }
>+> for (Block b : toInvalidate) {           //对于需要在 DataNode 上加以注销的每一个块
>+>+ blockLog.info("BLOCK * processReport: " + b + " on " + node + ...
                                                + " does not belong to any file")
>+>+ addToInvalidates(b, node)
>+> }
>+> for (BlockToMarkCorrupt b : toCorrupt) {   //对于需要被标注为损坏的每一个块
>+>+ markBlockAsCorrupt(b, storageInfo, node)
>+> }
> } //end else
> boolean staleBefore = storageInfo.areBlockContentsStale()
                                //该 storage 设备在此之前是否处于停运状态
> storageInfo.receivedBlockReport() //收到了本次报告,原来停运的 storage 可能已经恢复
>> if (heartbeatedSinceFailover) blockContentsStale = false
                                //如果从那之后收到过心跳,又有本次报告,就认为不再停运
>> blockReportCount ++
> if (staleBefore && !storageInfo.areBlockContentsStale()) { //如果曾经停运而现已恢复
>+ rescanPostponedMisreplicatedBlocks() //那就要重新扫描这个推迟处理的队列
> }
> return !node.hasStaleStorages() //返回 true 表示没有处于停滞状态的存储器

```

参数 nodeID 和 storage 唯一地确定了某个 DataNode 上的一个具体的存储设备,比方说磁盘或 RamDisk,第三个参数则是来自这个 DataNode、关于这个 storage 的报告。这里核心的操作其实只有两个:一个是 processReport(),这里已将其就地展开(注意函数名也叫 processReport,只是参数表不同);另一个是有条件的 rescanPostponedMisreplicatedBlocks(),即如果这个存储设备曾经停运但现已恢复就重新扫描一下这个被推迟的 List。至于 processFirstBlockReport(),那只是针对新增存储设备的优化,实质上与 processReport()一样。从展开了的摘要中可以看出,processReport()操作的核心是 reportDiff(),它从报告中归纳出这一次多了哪些块、移动了哪些块、应该作废哪些块、哪些块还在构建之中,等等,并将代表着这些块的数据结构(对象)分别放入 toAdd、toRemove、toInvalidate、toCorrupt、toUC 等集合中。UC 是“Under Construction”的缩写。一个块,如果还没有写满,也没有被关闭,它就是正在构建中,就是 UC。所以,processReport()得要在 reportDiff()完成了这些操作之后,才能对这几个集合分别进行诸如 addStoredBlock()等的操作。

```

[NameNodeRpcServer.blockReport() > BlockManager.processReport() > processReport()
> reportDiff()]

```

```

BlockManager.reportDiff (DatanodeStorageInfo storageInfo, BlockListAsLongs newReport,
    Collection<BlockInfo> toAdd, Collection<Block> toRemove,
    Collection<Block> toInvalidate, Collection<BlockToMarkCorrupt> toCorrupt,
    Collection<StatefulBlockInfo> toUC)

```

```

> BlockInfo delimiter = new BlockInfo(new Block(), 1) //用作已报告和未报告之间的分隔
> boolean added = storageInfo.addBlock(delimiter)
                //暂将此分隔对象加入 storageInfo 的队列前端
> headIndex = 0; //currently the delimiter is in the head of the list
> if (newReport == null) newReport = new BlockListAsLongs() //没有报告就生成一个空的
> // scan the report and process newly reported blocks
> BlockReportIterator itBR = newReport.getBlockReportIterator()
> while(itBR.hasNext()) {           //对于报告中的每一个块
>+ Block iblk = itBR.next()           //从报告中取得这个块的数据结构
>+ ReplicaState iState = itBR.getCurrentReplicaState()
                //可以是 FINALIZED、RBW、RWR、RUR 或 TEMPORARY
>+ BlockInfo storedBlock = processReportedBlock(storageInfo, iblk, iState,
                toAdd, toInvalidate, toCorrupt, toUC) //将其归类
>+ // move block to the head of the list
>+ if (storedBlock != null && (curIndex = storedBlock.findStorageInfo(storageInfo)) >= 0) {
>++ headIndex = storageInfo.moveBlockToHead(storedBlock, curIndex, headIndex)
                // 这个块原来就在目标存储器的 storageInfo 中(并非新出现的块)
                // 将这个块的数据结构移到队列前端,那一定是在 delimiter 之前
>+ }
>+ // 如果 storedBlock 是 null,就说明这个 iblk 是新出现的块,应该已归入 toAdd 中
> } //end while
> // collect blocks that have not been reported, all of them are next to the delimiter
> // 至此,如果 delimiter 后面还有未被移动的块,那就是原来有而现已不再存在的块了
> Iterator<BlockInfo> it = storageInfo.new BlockIterator(delimiter.getNext(0))
> while(it.hasNext()) toRemove.add(it.next()) //把这些块加到 toRemove 集合中
> storageInfo.removeBlock(delimiter) //最后把临时加入 storageInfo 队列中的 delimiter 去掉

```

每一个已登记 DataNode 上的每个存储设备,在 NameNode 上都有一个对应的 DatanodeStorageInfo 对象,即 storageInfo。这个 storageInfo 内部有个队列,存储在这个存储设备上的每一个块在这队列中都有个对应的 BlockInfo 对象。不过 DataNode 的块报告中只有 Block 对象,比 BlockInfo 小很多,因为 DataNode 只是被动地存储数据块,它所掌握的相关信息比 NameNode 少。不过一般而言根据 Block 对象中的块号,就可以在 storageInfo 中找到相应的 BlockInfo 对象。如果找不到,就说明这个 Block 是新的,以前没有,那就应该添加进去。反过来,如果 storageInfo 中有某个块的 BlockInfo 对象,而报告中却没有相应的 Block 对象,那就说明这个 DataNode 的这个存储设备上已经不再有这个块了,那就应该从代表着这个特定节点、特定设备的 storageInfo 中删去。这就是大概的思路。当然,这里所说的“块”实际上只是这个块的一个复份。

另外还要说明,以新增的块为例,也并非只需为其创建一个 BlockInfo 对象加到 storageInfo 的队列中就行。实际上并非那么简单,代表着具体 DataNode 的 DatanodeDescriptor 对象中还有别的信息也与之有关。所以只能先记下一笔账,等比对完了之后再来处理,而 toAdd 这个集

合就是用来记这个账的。同样的道理, toRemove 这个集合也是这样。还有 toInvalidate 集合, 则用来记下需要通知 DataNode 加以废除的块。

实际的比对是由 processReportedBlock() 完成的。如果根据给定的 Block 对象 iblk 在 storageInfo 中找到了对应的 BlockInfo, 这个 BlockInfo 对象就成为 storedBlock。

```
[NameNodeRpcServer.blockReport() > BlockManager.processReport() > processReport()
> reportDiff() > processReportedBlock()]
```

```
processReportedBlock(DatanodeStorageInfo storageInfo, Block block,
    ReplicaState reportedState, Collection<BlockInfo> toAdd, ...)
> DatanodeDescriptor dn = storageInfo.getDatanodeDescriptor()
> if (shouldPostponeBlocksFromFuture && namesystem.isGenStampInFuture(block)) {
>+ queueReportedBlock(storageInfo, block,
    reportedState, QUEUE_REASON_FUTURE_GENSTAMP)
>+ return null
> }
> // find block by blockId
> BlockInfo storedBlock = blocksMap.getStoredBlock(block)
    //从 BlockManager 的 blocksMap 中寻找 block 所对应的 BlockInfo 对象
> if(storedBlock == null) { //如果没有
>+ // If blocksMap does not contain reported block id, the replica
    should be removed from the data - node.
>+ toInvalidate.add(new Block(block))
    //既然 NameNode 上无记录,就准备要求 DataNode 将其撤销
>+ return null
> } //end if
> //找到了 NameNode 上与 block 对应的 BlockInfo 对象 storedBlock
> BlockUCState ucState = storedBlock.getBlockUCState()
> // Ignore replicas already scheduled to be removed from the DN
> if(invalidateBlocks.contains(dn, block)) return storedBlock
    //这个块已经在 invalidateBlocks 中等待被撤销,就不往下检查分类了
> BlockToMarkCorrupt c = checkReplicaCorrupt(block, reportedState,
    storedBlock, ucState, dn)
> if (c != null) { //这个块已经损坏
>+ if (shouldPostponeBlocksFromFuture) { //留待以后再定是否损坏
>++ queueReportedBlock(storageInfo, storedBlock,
    reportedState, QUEUE_REASON_CORRUPT_STATE)
>+ } else { //不留待以后了,直接进入 toCorrupt 集合
>++ toCorrupt.add(c) //这是已经损坏的块(复份)
>+ }
>+ return storedBlock
```

```

> } //end if
> if (isBlockUnderConstruction(storedBlock, ucState, reportedState)) { //是正在构建中的块
>+ ucb = new StatefulBlockInfo((BlockInfoUnderConstruction) storedBlock,
                                new Block(block), reportedState)
>+ toUC.add(ucb) //需要添加到正在构建的集合中
>+ return storedBlock
> } //end if
> // Add replica if appropriate. If the replica was previously corrupt but now okay,
                                it might need to be updated.
> if (reportedState == ReplicaState.FINALIZED &&
    (storedBlock.findStorageInfo(storageInfo) == -1
    || corruptReplicas.isReplicaCorrupt(storedBlock, dn))) /* 或认为已经损坏 */ {
    //如果 DataNode 上已 FINALIZED,却没有出现在 storageInfo 的
>+ toAdd.add(storedBlock)
> } //end if
> return storedBlock

```

等程序从 processReportedBlock() 返回到 reportDiff() 时, 如果这个块原先不在 BlockManager 的 blocksMap 中, 就返回 null; 这个块是 BlockManager 不认的, 所以已被加入到 toInvalidate 这个集合中准备予以撤销, 除非因 shouldPostponeBlocksFromFuture 为 true 而被搁置到以后再判定。反之, 如果这个块原来就在 blocksMap 中, 就返回对相应 BlockInfo 对象 storedBlock 的引用; 同时视具体情况可能已把这个 BlockInfo 对象加到了 toCorrupt、toUC 或 toAdd 中。

然后, 我们在前面看到, 每个存储设备的 blockList 维持着一个 BlockInfo 对象队列 (这些块都在这个设备上存储了一个复份); reportDiff() 会把这个 BlockInfo 对象移到这个队列的最前面。这样, 对于 DataNode 所报告的存储在这个设备上的所有的块进行了一遍扫描之后, 如果这个设备的 blockList 中还有些 BlockInfo 对象没有被移动, 那么这些块在那个设备上其实是不存在的, 应该从这个队列中去除, 所以 reportDiff() 会把这些 BlockInfo 对象放入 toRemove 集合。

这样, 当程序返回到 processReport() (带两个参数的 processReport()) 的时候, 就可以对这些集合分别采取措施了。比方说, 对于 toInvalidate 这个集合, 就是对其中的每个元素执行 addToInvalidates():

```
[NameNodeRpcServer.blockReport() > BlockManager.processReport() > addToInvalidates()]
```

```

addToInvalidates(Block block, DatanodeInfo datanode)
> if (!namesystem.isPopulatingReplQueues()) return
> invalidateBlocks.add(block, datanode, true)

```

这很简单, 就是把这个块的 Block 对象及其所在 DataNode 的 DatanodeInfo 添加进 BlockManager 的 invalidateBlocks 集合。于是, 当 NameNode 受到来自该 DataNode 的心跳时, 就会针对这些 Block 生成一个 DNA_INVALIDATE 命令。

相比之下对 toAdd 集合的处理就复杂多了,那是 addStoredBlock()的事:

```
[NameNodeRpcServer.blockReport() > BlockManager.processReport() > addStoredBlock()]
```

```
addStoredBlock(BlockInfo block, DatanodeStorageInfo storageInfo,
                DatanodeDescriptor delNodeHint, boolean logEveryBlock)
> node = storageInfo.getDatanodeDescriptor() //这个存储设备所在的 DataNode
> if (block instanceof BlockInfoUnderConstruction) {
>+ storedBlock = blocksMap.getStoredBlock(block)
> } else {
>+ storedBlock = block
> }
> if (storedBlock == null || storedBlock.getBlockCollection() == null) return block
> BlockCollection bc = storedBlock.getBlockCollection() //获知这个块所属的 HDFS 文件
> assert bc != null : "Block must belong to a file"
> boolean added = storageInfo.addBlock(storedBlock) //add block to the datanode,
                //注意:所谓加到这个 DataNode,是指加到关于这个 DataNode 的记载中
> if (added) {
>+ curReplicaDelta = 1
>+ if (logEveryBlock) logAddStoredBlock(storedBlock, node)
> } else { //如果加不进去就一定是有问题
>+ corruptReplicas.removeFromCorruptReplicasMap(block, node,
                Reason.GENSTAMP_MISMATCH)
>+ curReplicaDelta = 0
> }
> NumberReplicas num = countNodes(storedBlock) //数一下有几个节点存有其复份
> int numLiveReplicas = num.liveReplicas() //这是其中已经可用的复份数量
> numCurrentReplica = numLiveReplicas + pendingReplications.getNumReplicas(storedBlock)
                //加上已在复制计划中的复份数量
> if (storedBlock.getBlockUCState() == BlockUCState.COMMITTED
                && numLiveReplicas >= minReplication) {
                //对尚处于 UC 阶段的块,其复份数量只要够上最低要求就行
>+ storedBlock = completeBlock(bc, storedBlock, false)
> } else if (storedBlock.isComplete() && added) {
>+ namesystem.incrementSafeBlockCount(numCurrentReplica)
> }
> // if file is under construction, then done for now
> if (bc.isUnderConstruction()) return storedBlock
> // do not try to handle over/under-replicated blocks during first safe mode
> if (!namesystem.isPopulatingReplQueues()) return storedBlock
> // handle underReplication/overReplication,下面就来处理复份不足和过量的情况了
```

```

> short fileReplication = bc.getBlockReplication() //为这具体 HDFS 文件设置的复份数量
> if (!isNeededReplication(storedBlock, fileReplication, numCurrentReplica)) {
    //如果算下来已经不需要再有更多复份,就从 neededReplications 队列中 remove
>+ neededReplications.remove(storedBlock, numCurrentReplica,
                               num.decommissionedReplicas(), fileReplication)
> } else { //否则这是需要的
>+ updateNeededReplications(storedBlock, curReplicaDelta, 0)
> }
> if (numCurrentReplica > fileReplication) { //复份的数量太多了
>+ processOverReplicatedBlock(storedBlock, fileReplication, node, delNodeHint)
> }
> // If the file replication has reached desired value we can remove
    // any corrupt replicas the block may have. 如果复份数量已达标,但有坏块,就去掉
> corruptReplicasCount = corruptReplicas.numCorruptReplicas(storedBlock)
> numCorruptNodes = num.corruptReplicas()
> if ((corruptReplicasCount > 0) && (numLiveReplicas >= fileReplication))
>+ invalidateCorruptReplicas(storedBlock) //将其废除
> return storedBlock

```

总之,要加进一个块的复份也不是随便可加的,HDFS 对于复份数量有个控制,少了就补,多了就删。具体的 HDFS 文件还可单独设置复份数量。还要考虑也许已经安排了复制但是还在路上,从而 DataNode 的报告中还没有包括进去。这里所谓的加进一个块,是指准备加到 NameNode 的记载中为 NameNode 所接受的意思。既然 DataNode 的报告中有这个块,就是其本地文件系统中已经有了这个块的文件,这个块已经存在于那个节点上了。

最后,返回到前面带三个参数的那个 BlockManager.processReport()中,还要处理一种情况,就是如果一个存储设备原来曾处于停运状态,而现在已经恢复,那就要扫描一下 BlockManager 用于推迟处理的那个队列 postponedMisreplicatedBlocks。从代码中看,这主要是用来对付 NameNode 备份切换的。

```

[NameNodeRpcServer.blockReport() > BlockManager.processReport()
> rescanPostponedMisreplicatedBlocks()]

```

rescanPostponedMisreplicatedBlocks()

```

> for (Iterator<Block> it = postponedMisreplicatedBlocks.iterator(); it.hasNext();) {
    //对推迟处理队列中的每一个块
>+ Block b = it.next()
>+ BlockInfo bi = blocksMap.getStoredBlock(b)
>+ if (bi == null) { //如果 blocksMap 中没有这个块
>++ it.remove() //此项无效,把它从队列中去掉
>++ postponedMisreplicatedBlocksCount.decrementAndGet() //减少计数
>++ continue

```

```

>+ }
>+ MisReplicationResult res = processMisReplicatedBlock(bi) //确有其事,需要处理
>+ if (res!= MisReplicationResult.POSTPONE) { //如果不再延迟
>++ it.remove()
>++ postponedMisreplicatedBlocksCount.decrementAndGet()
>+ }
> }//end for

```

所谓 MisReplicated,就是复份的数量不对,所以需要或增补或删除。

```

[NameNodeRpcServer.blockReport() > BlockManager.processReport()
> rescanPostponedMisreplicatedBlocks() > processMisReplicatedBlock()]

```

processMisReplicatedBlock(BlockInfo block)

```

> BlockCollection bc = block.getBlockCollection()
> if (bc == null) { //block does not belong to any file
>+ addToInvalidates(block)
>+ return MisReplicationResult.INVALID
> }
> if (!block.isComplete()) {
>+ return MisReplicationResult.UNDER_CONSTRUCTION
> }
> // calculate current replication
> expectedReplication = bc.getBlockReplication()
> NumberReplicas num = countNodes(block)
> int numCurrentReplica = num.liveReplicas()
> // add to under - replicated queue if need to be
> if (isNeededReplication(block, expectedReplication, numCurrentReplica)) {
>+ if (neededReplications.add(block, numCurrentReplica,
                                num.decommissionedReplicas(), expectedReplication)) {
>++ return MisReplicationResult.UNDER_REPLICATED
>+ } // end if
> } // end if
> if (numCurrentReplica > expectedReplication) {
>+ if (num.replicasOnStaleNodes() > 0) {
>++ return MisReplicationResult.POSTPONE
>+ }
>+ // over - replicated block
>+ processOverReplicatedBlock(block, expectedReplication, null, null)
>+ return MisReplicationResult.OVER_REPLICATED
> }

```

```
> return MisReplicationResult.OK
```

我们在前面看到, NameNode 在对于心跳的处理中, 如果对方 DatanodeDescriptor 里面的队列 replicateBlocks 非空, 就说明有数据块需要补充复份, 要从这个 DataNode 加以复制, 因而 NameNode 会向这 DataNode 发出 DNA_TRANSFER 的命令。可是这个 replicateBlocks 队列中的内容又是怎么来的呢? 是谁把 BlockTargetPair 对象挂到这个队列中, 又怎么决定应该把新增的复份放在哪一个节点上?

我们在前面看到, 块管理器 BlockManager 内部有几个集合或队列, 还有一个复份监督线程 ReplicationMonitor。可想而知正是这个线程专门监督和控制着数据块复份的消长。为了解对于数据块复份的监督和控制, 我们重温一下 BlockManager 的摘要:

```
class BlockManager{
] BlocksMap blocksMap      //Mapping: Block -> { BlockCollection, datanodes, self ref }
] CorruptReplicasMap corruptReplicas  //已经损坏的复份(块扫描可以发现损坏)
      //Store blocks -> datanodedescriptor(s) map of corrupt replicas
] InvalidateBlocks invalidateBlocks    //Blocks to be invalidated., 需要撤销的复份
] Map<String, LightweightLinkedSet<Block>> excessReplicateMap  //过剩/多余的复份
      //Maps a StorageID to the set of blocks that are "extra" for this DataNode.
] UnderReplicatedBlocks neededReplications  //需要增加的复份, 按优先级的数组队列
      //Blocks that need to be replicated 1 or more times
] PendingReplicationBlocks pendingReplications  //等待增添的复份
] Daemon replicationThread = new Daemon(new ReplicationMonitor()) //监督线程
] class ReplicationMonitor implements Runnable {}  //监督线程的类型定义
]] run()
    > while (namesystem.isRunning()) {  //监督线程的主循环
    >+ if (namesystem.isPopulatingReplQueues()) {
    >++ computeDatanodeWork()
    >++ processPendingReplications()
    >+ }
    >+ Thread.sleep(replicationRecheckInterval)
    > }
]] computeDatanodeWork()
]] clearQueues()
    > neededReplications.clear()
    > pendingReplications.clear()
    > excessReplicateMap.clear()
    > invalidateBlocks.clear()
    > datanodeManager.clearPendingQueues()
]] static class ReplicationWork {}
```

NameNode 的 FSNamesystem 解决了从文件到数据块集合的映射, 使我们知道一个具体的文件包含了哪些数据块, 以及这些数据块的顺序, 也知道每个数据块应该有几个复份。但是

单凭持久存储在 NameNode 上的这些信息并不足以知道一个具体的数据块当前究竟有几个复份,以及每个复份的所在,这些信息有待于来自 DataNode 的报告。所以每个 DataNode 都要周期地向 NameNode 发出报告,里面就有关于本节点上存储着哪些块的复份及其存储类型的报告。根据这些报告,NameNode 就可以知道当前整个 HDFS 文件系统的实际情况。注意,这个实际情况是在动态变化的,一个数千节点的集群,难免一会儿这几个节点下去了,一会儿那几个节点又出问题了,再过一会儿又有几个节点回来了,还可能这个节点上的磁盘被拆装到了另一个节点上,这个节点的机器被移到了另一个机架上,或者 DataNode 上的块扫描发现某个数据块复份的文件损坏了。所以,NameNode 要根据来自 DataNode 的报告才能知道某个具体的数据块当前究竟有几个复份是真实可用的,具体又在哪几个节点上,是在磁盘上还是在 SSD 或 RamDisk 上,或者在缓存中。

我们假定一个数据块应该有三个复份(这是可以设置的,但默认为 3),可是 NameNode 可能发现某个数据块现在实际上只有两个复份了,那就需要补上一个,于是就需要从中选择一个复份,让其所在的节点将此复份拷贝到另一个节点上,使这个数据块的复份个数又回到 3。那么有没有可能一个数据块只剩下一个复份了,甚至一个也没有了呢?为说明问题,我们姑且假定丢失一个复份的概率是千分之一(实际上远小于此),那么同时丢失两个的概率就只有百万分之一了,发生这种情况的概率已是微乎其微;但只要还有一个留下,就仍旧可以拷贝出两个复份来。至于三个复份同时损坏的概率,那就只有十亿分之一,可以不考虑了。反过来,也有可能一个数据块突然有了四个复份,那是因为原先认为已经丢失的复份(或节点)又回来了,这时候就要废除其中的一个复份,以免占用太多的存储空间。

这显然应该是 BlockManager 的事,具体又是其内部 ReplicationMonitor 线程的事。我们在上面的摘要中看到,这个线程的主循环就是调用两个函数,然后睡眠,其中的第一个函数是 computeDatanodeWork(),就是要算一下该让哪些 DataNode 干点什么事:

```
[BlockManager.ReplicationMonitor.run() > computeDatanodeWork()]
```

```
BlockManager.ReplicationMonitor.computeDatanodeWork()
> if (namesystem.isInSafeMode()) return 0 //SafeMode 下不存在这个问题
> numlive = heartbeatManager.getLiveDatanodeCount() //有几个活着(有心跳)的 DataNode
> blocksToProcess = numlive * this.blocksReplWorkMultiplier //本次预计处理多少个数据块
> nodesToProcess = (int) Math.ceil(numlive * this.blocksInvalidateWorkPct) //处理多少个节点
> workFound = this.computeReplicationWork(blocksToProcess) //计算要复制哪些数据块
>> List<List<Block>>> blocksToReplicate = /* 按预计从中选取一组数据块加以计算 */
    neededReplications.chooseUnderReplicatedBlocks(blocksToProcess)
>> return computeReplicationWorkForBlocks(blocksToReplicate) //对这组数据块进行计算
> this.updateState()
> this.scheduledReplicationBlocksCount = workFound
> workFound += this.computeInvalidateWork(nodesToProcess) //计算要废除哪些复份
```

ReplicationMonitor 线程并非每次都计算所有的数据块和节点,而是在每个节点上抽取一定数量的数据块进行关于复制(复份)的计算,对一定百分比的节点进行关于废除(复份)的计算。这里的所谓“计算”实际上是调度、安排的意思。关于复制的计算是由 computeReplicationWork()

完成的,这里已经就地展开。这个函数先从队列 `neededReplications` 中按优先级从高到低摘取一定数量的 `Block` 对象,成为一个二维链表 `blocksToReplicate`,这些就是从 `DataNode` 的块报告中得到第一印象可能需要加以复制的数据块。下面就通过 `computeReplicationWorkForBlocks()` 逐块计算是否真的应该复制以及应该如何复制。

```
BlockManager.computeReplicationWorkForBlocks(List<List<Block>> blocksToReplicate)
> List<DatanodeDescriptor> containingNodes //含有这个块的复份的 DataNode 链表
> DatanodeDescriptor srcNode //用作复制源头的 DataNode
> List<ReplicationWork> work = new LinkedList<ReplicationWork>() // ReplicationWork 链表
> for (int priority=0; priority < blocksToReplicate.size(); priority++) { //对于每个优先级别
>+ for (Block block ; blocksToReplicate.get(priority)) {
//对于 blocksToReplicate 链表中属于这个优先级的每个块
>+ // block should belong to a file
>+ bc = blocksMap.getBlockCollection(block) //这个块所属的 HDFS 文件
>+ // abandoned block or block reopened for append
>+ if(bc == null || (bc.isUnderConstruction() && block.equals(bc.getLastBlock())) {
//如果这是个“无主”的块,或者是尚未关闭的文件中的最后一块,那就不用理睬
>+ neededReplications.remove(block, priority); // remove from neededReplications,撤销要求
>+ neededReplications.decrementReplicationIndex(priority)
>+ continue
>+ }
>+ requiredReplication = bc.getBlockReplication() //为这 HDFS 文件设置的复份数量
>+ // get a source data - node
>+ containingNodes = new ArrayList<DatanodeDescriptor>() //创建这个链表,见上
>+ List<DatanodeStorageInfo> liveReplicaNodes = new ArrayList<DatanodeStorageInfo>()
>+ numReplicas = new NumberReplicas()
>+ srcNode = chooseSourceDatanode(block, containingNodes,
//找一个 DataNode 作为复制该块复份的源头
liveReplicaNodes, numReplicas, priority)
>+ if(srcNode == null) { // block can not be replicated from any node
>+ LOG.debug("Block " + block + " cannot be repl from any node")
>+ continue //竟然没有一个 DataNode 可以作为源头,应该不会发生
>+ }
>+ // liveReplicaNodes can include READ_ONLY_SHARED replicas
//which are not included in the numReplicas.liveReplicas() count
>+ assert liveReplicaNodes.size() >= numReplicas.liveReplicas()
>+ // do not schedule more if enough replicas is already pending
>+ numEffectiveReplicas = numReplicas.liveReplicas() +
pendingReplications.getNumReplicas(block)
//有效复份的数量应包含已经安排但尚未完成复制的那些复份
>+ if (numEffectiveReplicas >= requiredReplication) { //有效复份的数量已满足要求
```

```

>+++ if ((pendingReplications.getNumReplicas(block) > 0) || (blockHasEnoughRacks(block))) {
>++++ neededReplications.remove(block, priority); // remove from neededReplications
>++++ neededReplications.decrementReplicationIndex(priority)
>++++ blockLog.info("BLOCK * Removing " + block
                    + " from neededReplications as it has enough replicas")
>++++ continue //已撤销要求
>+++ }
>++ }

    //该撤销要求的已经撤销,到了这里就至少要补充一个备份,也许更多
>++ if (numReplicas.liveReplicas() < requiredReplication) { //备份数量不足
>+++ additionalReplRequired = requiredReplication - numEffectiveReplicas //计算应补数量
>++ } else {
>+++ additionalReplRequired = 1; // Needed on a new rack
>++ }
>++ w = new ReplicationWork(block, bc, srcNode, containingNodes, liveReplicaNodes,
                        additionalReplRequired, priority) //创建一个 ReplicationWork 对象
>++ work.add(w) //并将其加入 work 集合,这是一个 ReplicationWork 链表,见上
>+ } //end for
> } //end for

    //现在我们有了一个 ReplicationWork 的链表,下面分两次扫描这个链表
    //根据这些 ReplicationWork 决定将某个块挂入 BlockManager 的某个队列
> Set<Node> excludedNodes = new HashSet<Node>()
> for(ReplicationWork rw : work) { //对于这个 work 链表中的每个 ReplicationWork
>+ excludedNodes.clear()
>+ for (DatanodeDescriptor dn : rw.containingNodes) excludedNodes.add(dn)
                        //已经有这个 block 的备份存在的那些 DataNode 应予排除
>+ rw.chooseTargets(blockplacement, storagePolicySuite, excludedNodes)
                        //挑选新增备份的去处
>+> targets = blockplacement.chooseTarget(bc.getName(), additionalReplRequired, srcNode,
                        liveReplicaStorages, false, excludedNodes, block.getNumBytes(),
                        storagePolicySuite.getPolicy(bc.getStoragePolicyID()))
                        == BlockPlacementPolicyDefault.chooseTarget(...)
> }
> for(ReplicationWork rw : work) { //第二次扫描 work 链表
>+ DatanodeStorageInfo[] targets = rw.targets //上一遍扫描中已经选择好目标节点
>+ if(targets == null || targets.length == 0) { //如果没能选出目标就只好算了
>++ rw.targets = null
>++ continue
>+ }
>+ Block block = rw.block //这个 ReplicationWork 所处理的块

```

```

>+ priority = rw.priority
>+ bc = blocksMap.getBlockCollection(block) //这个块所属的 HDFS 文件
    //Recheck since global lock was released block should belong to a file
>+ if(bc == null || (bc.isUnderConstruction() && block.equals(bc.getLastBlock()))){
    //如果不属于任何 HDFS 文件,或者是未关闭文件的最后一块,那就不用复制
>++ neededReplications.remove(block, priority); // remove from neededReplications
>++ rw.targets = null
>++ neededReplications.decrementReplicationIndex(priority)
>++ continue
>+ }

>+ requiredReplication = bc.getBlockReplication() //这个 HDFS 文件所要求的复份数量
>+ NumberReplicas numReplicas = countNodes(block) //已有几个复份
    //do not schedule more if enough replicas is already pending
>+ numEffectiveReplicas = numReplicas.liveReplicas() +
    pendingReplications.getNumReplicas(block)
>+ if (numEffectiveReplicas >= requiredReplication) {
    //有效复份数量已经够了,无须复制
>++ if( (pendingReplications.getNumReplicas(block) > 0) || (blockHasEnoughRacks(block)) ) {
>+++ neededReplications.remove(block, priority); // remove from neededReplications
>+++ neededReplications.decrementReplicationIndex(priority)
>++++ rw.targets = null
>++++ continue
>++ }
>+ }

>+ if( (numReplicas.liveReplicas() >= requiredReplication) &&
    (!blockHasEnoughRacks(block)) ) {
>++ if (rw.srcNode.getNetworkLocation().equals(
    targets[0].getDatanodeDescriptor().getNetworkLocation())) continue
    //如果所选目标与源头在同一机架上,那就意义不大,跳过
>+ }

>+ rw.srcNode.addBlockToBeReplicated(block, targets) //Add block to the to be replicated list
>++ replicateBlocks.offer(new BlockTargetPair(block, targets))
    //为这个 block 创建一个 BlockTargetPair,挂入 replicateBlocks 队列,请求复制
>+ scheduledWork ++ //计数
>+ DatanodeStorageInfo.incrementBlocksScheduled(targets)
>+ pendingReplications.increment(block, DatanodeStorageInfo.toDatanodeDescriptors(targets))
>+ if(numEffectiveReplicas + targets.length >= requiredReplication) {
    //现有的有效复份,加上这一次请求复制的,如果已经够了
>++ neededReplications.remove(block, priority); // remove from neededReplications
    //从 neededReplications 队列中去掉这个块

```

```

> ++ neededReplications.decrementReplicationIndex(priority)
> + }
> } //end for(ReplicationWork rw : work), 第二次扫描 work 链表结束
> return scheduledWork

```

这个 replicateBlocks 队列中的那些 BlockTargetPair, 就是这样来的。

这里还有一个 BlockPlacementPolicyDefault.chooseTarget(), 显然是根据某种策略选择复制数据块复份的目标节点, 而 BlockPlacementPolicyDefault 是默认采用的策略, 基本的考虑是尽量不要放在同一个机架(Rack)上。我们就不深入下去了。

下面是 processPendingReplications(), 这是对已经安排但尚未完成的复制工作的处理, 因为有可能已经超时:

```

[BlockManager.ReplicationMonitor.run() > BlockManager.processPendingReplications()]

// If there were any replication requests that timed out, reap them
// and put them back into the neededReplication queue
BlockManager.processPendingReplications()
> Block[] timedOutItems = pendingReplications.getTimedOutBlocks()
> if (timedOutItems != null) {
> + for (int i = 0; i < timedOutItems.length; i++) {
> ++ NumberReplicas num = countNodes(timedOutItems[i])
> ++ if (isNeededReplication(timedOutItems[i],
> ++ getReplication(timedOutItems[i]), num.liveReplicas())) {
> +++ neededReplications.add(timedOutItems[i], num.liveReplicas(),
> ++ num.decommissionedReplicas(), getReplication(timedOutItems[i]))
> ++ } //end if
> + } //end for
> } //end if

```

考察了 NameNode 对于心跳和 blockReport 的反应, 让我们回到 DataNode 这一侧。

NameNode 的响应报文上搭载着对于 DataNode 的命令, DataNode 需要加以执行。不过集群中可能有两个(两种)NameNode: 一是 Active, 即当前在位的 NameNode; 二是 Standby, 即处于候补状态的 NameNode。相应地, BPOfferService 提供了两个操作方法: 一个是 processCommandFromActive(); 另一个是 processCommandFromStandby()。当然, 候补的 NameNode 没有什么发言权, 正常情况下它也不会越权, 所以我们在这里只关心前者。

下面是这个函数的摘要:

```

[BPSERVICEActor.run() > offerService() > BPOfferService.processCommandFromActive()]

processCommandFromActive(DatanodeCommand cmd, BPSERVICEActor actor)
> BlockCommand bcmd = cmd instanceof BlockCommand?(BlockCommand)cmd; null
> BlockIdCommand blockIdCmd = cmd instanceof BlockIdCommand?

```

```
(BlockIdCommand)cmd: null
```

```
> switch(cmd.getAction()) {
> case DatanodeProtocol.DNA_TRANSFER: //将若干数据块发送到别的 DataNode
>+ dn.transferBlocks(bcmod.getBlockPoolId(), bcmod.getBlocks(),
                    bcmod.getTargets(), bcmod.getTargetStorageTypes())
>+ break
> case DatanodeProtocol.DNA_INVALIDATE: //将若干数据块作废
>+ Block toDelete[] = bcmod.getBlocks()
>+ if (dn.blockScanner!= null) { //从 BlockScanner 那里删除,让它不再扫描本块
>++ dn.blockScanner.deleteBlocks(bcmod.getBlockPoolId(), toDelete)
>+ }
>+ dn.getFSDataset().invalidate(bcmod.getBlockPoolId(), toDelete) //向 FsDatasetImpl 注销本块
>+ break
> case DatanodeProtocol.DNA_CACHE: //使若干数据块进入缓存,以加快读取
>+ dn.getFSDataset().cache(blockIdCmd.getBlockPoolId(), blockIdCmd.getBlockIds())
>+> for (int i = 0; i < blockIds.length; i++) {
>+>+ cacheBlock(bpid, blockIds[i])
>+>+> FsDatasetCache.cacheBlock(long blockId, String bpid, String blockFileName,
                                long length, long genstamp, Executor volumeExecutor)
>+>+>> volumeExecutor.execute(new CachingTask(key, blockFileName, length, genstamp))
                                //创建并执行 Runnable 线程 CachingTask
>+> }
>+ break
> case DatanodeProtocol.DNA_UNCACHE: //使若干数据块退出缓存
>+ dn.getFSDataset().uncache(blockIdCmd.getBlockPoolId(), blockIdCmd.getBlockIds())
>+ break
> case DatanodeProtocol.DNA_SHUTDOWN:
>+ throw new UnsupportedOperationException(
                                "Received unimplemented DNA_SHUTDOWN")
> case DatanodeProtocol.DNA_FINALIZE: //封存一个 BlockPool 中的数据块
>+ String bp = ((FinalizeCommand) cmd).getBlockPoolId()
>+ dn.finalizeUpgradeForPool(bp)
>+> storage.finalizeUpgrade(blockPoolId)
>+>> for (StorageDirectory sd : storageDirs) {
>+>>+ File prevDir = sd.getPreviousDir()
>+>>+ if (prevDir.exists()) {
>+>>+> doFinalize(sd) //data node level storage finalize
>+>>+>> File prevDir = sd.getPreviousDir()
>+>>+>> String dataDirPath = sd.getRoot().getCanonicalPath()
>+>>+>> assert sd.getCurrentDir().exists() : "Current directory must exist."

```



```

>+>>+> File tmpDir = sd.getFinalizedTmp();//finalized.tmp directory
>+>>+> File bbwDir = new File(sd.getRoot(), Storage.STORAGE_1_BBW)
>+>>+> rename(prevDir, tmpDir) // 1. rename previous to finalized.tmp
>+>>+> new Daemon(new Runnable()).start()
    ] run()
        > deleteDir(tmpDir)
        > if (bbwDir.exists()) deleteDir(bbwDir)
>+>>+> } else {
>+>>+> BlockPoolSliceStorage bpStorage = bpStorageMap.get(bpID)
>+>>+> bpStorage.doFinalize(sd.getCurrentDir())
>+>>+> }
>+>> } //end for
>+ break
> case DatanodeProtocol.DNA_RECOVERBLOCK: //恢复某些数据块的原状
>+ String who = "NameNode at " + actor.getNNSocketAddress()
>+ dn.recoverBlocks(who, ((BlockRecoveryCommand)cmd).getRecoveringBlocks())
>+ break
> case DatanodeProtocol.DNA_ACCESSKEYUPDATE: //添加密钥
>+ if (dn.isBlockTokenEnabled) {
>+ dn.blockPoolTokenSecretManager.addKeys(getBlockPoolId(),
    ((KeyUpdateCommand) cmd).getExportedKeys())
>+ }
>+ break
> case DatanodeProtocol.DNA_BALANCERBANDWIDTHUPDATE: //调整网络带宽
>+ long bandwidth = ((BalancerBandwidthCommand) cmd).getBalancerBandwidthValue()
>+ if (bandwidth > 0) {
>+ DataXceiverServer dxcs = (DataXceiverServer) dn.dataXceiverServer.getRunnable()
>+ dxcs.balanceThrottler.setBandwidth(bandwidth)
>+ }
>+ break
> default: LOG.warn("Unknown DatanodeCommand action: " + cmd.getAction())
> }

```

除 DNA_UNKNOWN 和 DNA_REGISTER 以外还有 9 种有效的 DNA,都在这里了。换言之,NameNode 可以要求 DataNode 去做的,就是这么 9 种。注意,这里没有读写数据块的 DNA,因为 NameNode 和 DataNode 之间没有这样的活动,读写数据块的要求都来自 App。

我们在这里最关心的是 DNA_TRANSFER,DataNode 因此而执行的操作 transferBlocks() 是把一组数据块发送到别的 DataNode 上去,在那里再留一个复份。这属于 DataNode 之间的交互(而不是 DataNode 与 NameNode 之间的交互),所以把它放在下一章。

第14章

DataNode 间的互动

14.1 数据块的接收和存储

NameNode 指定将一个块的若干复份 (replica) 分别存储在什么节点上,但是它本身不会把数据块发送给 DataNode。实际上 NameNode 根本就不过手任何数据块,它只是处理元数据。把数据块发送给 DataNode 的,是用户的 App 或 Shell,还有就是 DataNode 之间的互相发送。

DataNode 会在其初始化过程中的 startDataNode() 里面通过 initDataXceiver() 创建一个服务线程 DataXceiverServer,这个线程的作用就是等待网络上 TCP 连接请求的到来,有请求到来就另外创建一个线程 DataXceiver 与之对接,然后又继续等待别的连接请求到来。对网络技术有所了解的读者想必知道,这是典型的服务端运行模式。一个集群中 DataNode 的数量动不动就是几百几千,甚至上万,让每一个 DataNode 都长期保留对集群中每一个 DataNode 的连接是不现实的,只能是有需要时就建立点对点的连接,不再需要时就断开,就像互联网中那样。Xceiver 这个词意味着 Transmitter 和 Receiver 的合成。

下面是服务线程 DataXceiverServer 的摘要:

```
class DataXceiverServer implements Runnable{
    ] run()
    > while (datanode.shouldRun &&!datanode.shutdownForUpgrade) {
    >+ peer = peerServer.accept() //等待连接请求的到来并建立连接
        //peerServer 是个 TcpPeerServer,DataXceiverServer 的下层是 TCP/IP 传输层
    >+ new Daemon(datanode.threadGroup, DataXceiver.create(peer, datanode, this)).start()
        //创建 DataXceiver 线程并调用其 start() 函数
    > } //只要节点不关闭,就永远循环
    > peerServer.close()
    > closeAllPeers()
```

需要说明一下,作为 DataNode 的数据收发服务,与其连接的请求只能来自集群中别的数据块或 App,而不可能是 NameNode。NameNode 是永远不会主动来与 DataNode 连接的,这不仅是因为我们指定了 NameNode 为 Master 而 DataNode 都是 Slave,也是因为:集群中只有一个节点是 NameNode(即使在 Federal 模式下,NameNode 也只有少数几个),而 DataNode 则是大量的,要让众多 DataNode 知道谁是 NameNode 容易,反过来就难了。另一方面,后面我们会看到,数据块只是在 DataNode 之间,或者在 DataNode 与 App 之间流通,而

DataNode 与 NameNode 之间是不会有数据块流通的。NameNode 跟踪和管理数据块,但自己不“过手”数据块。

我们现在关心的是 DataNode 之间的数据块流通,把 DataNode 与 App 之间的互动放在后面。

这样,有连接请求到来时,DataXceiverServer 就创建一个 DataXceiver 线程与之对接,这就建立起了两个 DataNode 之间的数据通道。这个数据通道,从而这个线程,一般不会永久存在,只是在有需要时才创建,完成了一个几个数据块的传输之后就退出了,下次有需要就再次建立。读者也许会想,怎么建立一个连接就只传一个几个数据块?须知典型的数据块大小是 64MB,传上两个数据块就是 128MB 了。另外,如上所述,要让 DataNode 长期保留对每个 DataNode 的连接也不现实。不过事实上 DataNode 确实允许同时有很多个 DataXceiver 线程存在,这是可以在配置文件中的“dfs.datanode.max.transfer.threads”条目下设定的,默认值竟是 4096。另外,每个这样的连接可以占用多少网络带宽,每秒钟允许其传输多少数据,也是可以在配置文件中设定的,由一个 BlockBalanceThrottler 对象加以限制和调节。注意,DataXceiverServer 既然是 Server,就总是处于被动的地位,通信总是由对方发起的,DataXceiverServer 不会主动发起。当然,DataNode 之间是对等的,所以每个 DataNode 上必然还有发起连接的机制。

实际连接时才创建的 DataXceiver 线程,也即 DataXceiver 类对象,是真正的工作线程。简单地说,这个线程的作用就是从网络上接收来自其他 DataNode 的消息,解析其操作请求并加以执行。如果操作请求是 WRITE_BLOCK,就先接收这个数据块的元数据即关于数据的说明;然后分多次(因为 Packet 比 Block 小很多)接收其数据部分,作为文件写入本地的宿主文件系统,同时在内存中创建相关的数据结构(对象)。在宿主文件系统中创建数据块文件时,开始是在 rbw 目录下,正确完成了整个数据块的接收和写入之后再把它转入 finalized 目录下,并报告给 NameNode。

如前所述,为达到容错的目标,每个数据块并非只存储在一个 DataNode 上,而是有多个复份(一般是三个),分别存储在多个不同的 DataNode 上。要把一个数据块的多个复份存放在多个 DataNode 上,当然不是只有一种方法,HDFS 采用的是“接力”“流水线(Pipeline)”的方法,就是:发送者把几个目标节点排成一个列表,只把数据连同这个“路条”发给其中的第一个节点;第一个节点收到之后一方面当然要在本地存储,另一方面还要转发给第二个节点;其余类推。原始的发送者在“路条”中还指定了其中每一个节点的存储类型,即 DISK、SSD 或 RAMDISK,这些存储设备都挂在相关宿主机的文件系统中,但是对应着不同的访问路径。其中 RAMDISK 是挥发性的,容量也很有限,所以还专门有个线程负责按 LRU(最近最少访问)算法过一段时间就把其中的一些数据块文件转移到磁盘上。

下面我们就来考察其具体的实现。

首先,Receiver 是个抽象类,它实现了用于 DataNode 之间通信的 DataTransferProtocol 协议:

```
abstract class Receiver implements DataTransferProtocol {}  
] DataInputStream in      //Receiver 的输入流,这是 Receiver 下面的 TCP 传输层  
] readOp()                //从输入流里面读出操作码 Op  
] processOp(Op op)        //执行 Op 所规定的操作,根据 Op 调用不同的方法
```

```

> switch(op) {
> case READ_BLOCK:    opReadBlock()
> case WRITE_BLOCK:   opWriteBlock(in)
> case REPLACE_BLOCK: opReplaceBlock(in)
> case COPY_BLOCK:    opCopyBlock(in)
> case BLOCK_CHECKSUM: opBlockChecksum(in)
> case TRANSFER_BLOCK: opTransferBlock(in)
> ...
> }

] opReadBlock()           //从本节点读取一个数据块
> OpReadBlockProto proto = OpReadBlockProto.parseFrom(vintPrefixed(in))
> readBlock(...)

] opWriteBlock(DataInputStream in) //将一个数据块写入本节点以及流水线中的后续节点
> OpWriteBlockProto proto = OpWriteBlockProto.parseFrom(vintPrefixed(in))
> DatanodeInfo[] targets = PBHelper.convert(proto.getTargetsList())
> writeBlock(...)

] opTransferBlock(DataInputStream in) //将数据块复份发送到别的节点
> OpTransferBlockProto proto = OpTransferBlockProto.parseFrom(vintPrefixed(in))
> DatanodeInfo[] targets = PBHelper.convert(proto.getTargetsList())
> transferBlock(...)

] opReplaceBlock(DataInputStream in) //将复份换个地方,从一个节点换到另一个节点
> OpReplaceBlockProto proto = OpReplaceBlockProto.parseFrom(vintPrefixed(in))
> replaceBlock(...)

] opCopyBlock(DataInputStream in)      //拷贝一个数据块复份
] opBlockChecksum(DataInputStream in) //对一个数据块复份实施 Checksum 检查

```

Receiver 是建立在下层 TCP 传输层基础上的, Receiver 与对方之间的协议就是 DataTransferProtocol。在 DataTransferProtocol 报文的头部有个操作码 Op, 怎样解释报文的内容则取决于 Op 的值, 那又有另一层次的协议。例如当 Op 为 READ_BLOCK 的时候, 报文的正文就是一个 OpReadBlockProto。而方法 processOp(), 则根据 Op 的值而调用不同的方法, 来处理一个 DataTransferProtocol 报文的正文部分。以 opWriteBlock() 为例, 则从正文部分解析还原出一个 OpWriteBlockProto 报文, 再按此协议从中提取种种信息, 作为调用 writeBlock() 的参数。不过 Receiver 是个抽象类, 并未提供具体实现 writeBlock() 的代码, 而是留待扩充这个抽象类的具体类来加以实现。之所以如此, 是因为这样就可以有多个不同的具体类都来扩充这同一个抽象类, 以应对不同的具体情况。另外, Receiver 虽然提供了 readOp()、processOp() 等函数, 却只是相当于一组可供调用的库函数, Receiver 本身并不是一个线程, 没有 run() 函数来调用这些函数。

DataXceiver 就是对 Receiver 的扩充, 而且是个线程, 更确切地说是供线程执行的 Runnable 对象。

```
class DataXceiver extends Receiver implements Runnable {}
```

```

] Peer peer          //实现 Peer 界面的某类对象(如 BasicInetPeer),代表着与对方的连接
] String remoteAddress // address of remote side,对方的地址
] String localAddress  // local address of this daemon,本地的地址
] DataNode datanode    //这个节点上的 DataNode 对象
] DataXceiverServer dataXceiverServer //服务线程
] InputStream socketIn //由 socket 提供的输入流
] OutputStream socketOut //由 socket 提供的输出流
] BlockReceiver blockReceiver //数据块接收器
] sendOOB()            //Out-Of-Band 报文都是控制信息
    > blockReceiver.sendOOB() //通过 blockReceiver 发送 OOB 控制信息
] run()
    > dataXceiverServer.addPeer(peer, Thread.currentThread(), this) //peer 就是与对方的连接
    > InputStream input = socketIn //底层的 TCP 输入流
    > super.initialize(new DataInputStream(input)) //调用 Receiver.initialize()
    > do { //process requests in a loop
    >+ op = readOp() //从输入流中读取操作码 op
    >+ processOp(op) //调用 Receiver.processOp()
    > } while ((peer != null) && (!peer.isClosed() && dnConf.socketKeepaliveTimeout > 0))
    > dataXceiverServer.closePeer(peer)
] readBlock(final ExtendedBlock block, ...) //对方要求从本节点读一个块(复份)
] writeBlock(ExtendedBlock block, StorageType storageType, ...,
    DatanodeInfo[] targets, StorageType[] targetStorageTypes, ...)
    //要求将一个块写入本节点(留下一个复份),并向下游节点转发
] transferBlock(ExtendedBlock blk, ..., DatanodeInfo[] targets,
    StorageType[] targetStorageTypes)
    //向下游发送本节点上的一个块(复份),用于增添复份
] copyBlock(final ExtendedBlock block, Token<BlockTokenIdentifier> blockToken)
    //复制数据块,用于负载均衡。
] replaceBlock(final ExtendedBlock block, StorageType storageType, ...)
    //从一个节点转移一个数据块复份到当前节点

```

DataXceiver 一方面是对 Receiver 的扩充和落实,具体提供了 readBlock()、writeBlock() 等函数;一方面又按 Runnable 界面的规定提供了一个 run()函数,成为一个线程。

这样,线程 DataXceiver 就在一个 do{}while()循环中通过 readOp()从输入流中读取一个操作码 op,然后就通过 processOp()加以处理,处理完之后又从输入流中读取下一个操作码,暂时读不到就睡一会儿。这个循环要到什么时候才结束呢?从代码中可以看到,条件之一是 peer.isClosed(),就是通信的对方也即通信的发起方断开了与本节点的 TCP 连接。

DataXceiver 是对 Receiver 的扩充,这里对 processOp 的调用就是对 Receiver.processOp()的调用。反过来,Receiver.opWriteBlock()那里面调用的 writeBlock()则是 DataXceiver 的 writeBlock()。

不过 DataXceiver 终究只是对于 Receiver 的扩充, 尽管号称 Xceiver(一般指收发器), 它本身并不提供发送能力。但是它又确实需要发送, 否则就没法执行 readBlock(), 因为那是对方要求从这个 DataNode 上读取一个数据块, 你得把它发送过去。但是, 纵然如此, 这种发送也只是被动的发送, DataXceiver 作为 Receiver 不能主动发起节点间的交互, 它没有提供这样的操作方法。与 DataXceiver 对接、能够发起 DataTransferProtocol 通信的是发送器 Sender:

```
class Sender implements DataTransferProtocol {}

] DataOutputStream out                //这是一个基于 socket 的 TCP 输出流
] op(final DataOutput out, final Op op) //将操作码 op 写入输出流
    > out.writeShort(DataTransferProtocol.DATA_TRANSFER_VERSION) //首先是版本号
    > op.write(out)                //然后是操作码
] send(final DataOutputStream out, final Op opcode, Message proto) //发送一个操作请求
    > op(out, opcode)                //操作请求的尾部是版本号 + 操作码
    > proto.writeDelimitedTo(out)    //报文内容, 最后是一个分隔符
    > out.flush()                    //冲刷输出流
] readBlock(final ExtendedBlock blk, ..., CachingStrategy cachingStrategy)
    //发送 READ_BLOCK 请求报文
    > head = DataTransferProtoUtil.buildClientHeader(blk, clientName, blockToken)
    //报头的构成, 包括对数据块的描述等信息
    > proto = OpReadBlockProto.newBuilder() //构筑 OpReadBlockProto 报文
        .setHeader(head)                //首先是报头
        .setOffset(blockOffset)         //然后是块内的起点
        .setLen(length)                  //需要读取的长度
        .setSendChecksums(sendChecksum) //Checksum, 例如 CRC
        .setCachingStrategy(...).build() //是否将目标块置入缓存
    > send(out, Op.READ_BLOCK, proto)    //在报文末尾添上操作码并发送
] writeBlock(final ExtendedBlock blk, ..., boolean allowLazyPersist)
    > header = DataTransferProtoUtil.buildClientHeader(blk, clientName, blockToken)
    //报头的构成, 包括对数据块的描述等信息
    > proto = OpWriteBlockProto.newBuilder() //构筑 OpWriteBlockProto 报文
        .setHeader(header)                //首先是报头
        .setStorageType(PBHelper.convertStorageType(storageType))
        //然后是第一个节点的存储类型
        .addAllTargets(PBHelper.convert(targets, 1)) //流水线中的所有节点
        .addAllTargetStorageTypes(PBHelper.convertStorageTypes(targetStorageTypes, 1))
        //所有节点的存储类型
        .setStage(toProto(stage))          //本次数据块写入所处的阶段
        .setPipelineSize(pipelineSize)     //流水线长度
        ...                                //其他
        .setAllowLazyPersist(allowLazyPersist) //是否延迟进行持久化
    > proto.setSource(PBHelper.convertDatanodeInfo(source)) //产生数据的源节点
```



```

> send(out, Op.WRITE_BLOCK, proto.build())           //在报文末尾添上操作码并发送
] transferBlock(final ExtendedBlock blk, ..., StorageType[] targetStorageTypes)
> proto = OpTransferBlockProto.newBuilder()...build()
> send(out, Op.TRANSFER_BLOCK, proto)

```

可想而知,当一个节点想要发起与某一目标节点的数据传输时,它首先要跟目标节点建立连接并创建一个 Sender,然后调用这个 Sender 的 readBlock()或 writeBlock()。注意这两个方法虽然都说是 readBlock 和 writeBlock,但实际上不一定是整块的读写,而可以是数据块的一部分,可以是在数据块范围之内任意长度的读写,只是写操作只能是在原有数据之后的 append。

DataXceiver 之所谓只能被动接受来自上游的操作请求,其实只是针对上游节点而言,实际上它很可能还需要在一个流水线中起中继转发的作用;这时候,对于下游节点而言,它就是操作的发起者了。所以 DataXceiver 在整体上、宏观上一定是被动的,但是在局部的细节上却也可以是主动的,此时 DataXceiver 也要根据需要创建 Sender。

于是,当作为主动方的 DataNode 发起将一个数据块的 N 个复份写入以某个 DataNode 为首的 N 个数据节点时,首先当然要建立连接,然后通过 Sender 发送操作请求,报文中的操作码是 WRITE_BLOCK;而作为被动方的那个 DataNode 上与之对接的线程 DataXceiver,则通过 processOp()和 opWriteBlock()调用其 writeBlock()。这个 writeBlock()正是我们在这里首先要考察的操作过程(注意,这是在处于下游的接收方,即被动方):

```
[DataXceiver.run() > Receiver.processOp() > DataXceiver.writeBlock()]
```

```

DataXceiver.writeBlock (           //15 个调用参数!
    ExtendedBlock block,           //对于数据块本身的描述,包括 poolId
    StorageType storageType,       //本节点上的存储类型,如 DISK、RAMDISK
    Token<BlockTokenIdentifier> blockToken, //访问许可证
    String clientname,             //发起操作的用户
    DatanodeInfo[] targets,        //N-1 个用来存储数据块复份的 DataNode
    StorageType[] targetStorageTypes, //N-1 个数据块复份的存储类型
    DatanodeInfo srcDataNode,      //发起写操作的源节点
    BlockConstructionStage stage,  //目标块所处的构造阶段,详见正文说明
    int pipelineSize,              //流水线的长度,即 N-1(不包括本节点)
    long minBytesRcvd,             //接收数据量的最小值
    long maxBytesRcvd,             //接收数据量的最大值
    long latestGenerationStamp,    //最新的世代标记(版本号)
    DataChecksum requestedChecksum, //所需的 checksum 方式,如 CRC 等
    CachingStrategy cachingStrategy, //缓存策略(数据块复份是否进入缓存)
    boolean allowLazyPersist       //是否允许延迟持久化
)
> boolean isDatanode = clientname.length() == 0 //clientname 为空表示是由 DataNode 发起
> boolean isClient = !isDatanode                //否则就是由用户发起的(不会由 NN 发起)

```

```

> boolean isTransfer = (stage == BlockConstructionStage.TRANSFER_RBW
                        || stage == BlockConstructionStage.TRANSFER_FINALIZED)
> if (isTransfer && targets.length > 0) { //对 TRANSFER 不允许 pipelining
>+ throw new IOException(stage + " does not support multiple targets " + Arrays.asList(targets))
> }
> originalBlock = new ExtendedBlock(block) //将 ExtendedBlock 复制保留一个副本
> bos = new BufferedOutputStream(getOutputStream(),
                                HdfsConstants.SMALL_BUFFER_SIZE)
> replyOut = new DataOutputStream(bos) //创建用来发送应答报文的数据输出流
> //第一步,根据需要创建 BlockReceiver 对象
> if (isDatanode || stage != BlockConstructionStage.PIPELINE_CLOSE_RECOVERY) {
                                //这是正常的数据块传输阶段,需要创建 BlockReceiver 对象
>+ blockReceiver = new BlockReceiver(block, storageType, in,
                                peer.getRemoteAddressString(), ...)
>+ storageUuid = blockReceiver.getStorageUuid()
> } else { //这是 PIPELINE_CLOSE_RECOVERY 阶段,无须创建 BlockReceiver 对象
>+ storageUuid = datanode.data.recoverClose(block, latestGenerationStamp, minBytesRcvd)
> }
> //第二步,根据需要建立与下游节点的连接并转发
> if (targets.length > 0) { //Connect to downstream machine, if appropriate
>+ mirrorNode = targets[0].getXferAddr(connectToDnViaHostname) //本节点的下游节点
>+ mirrorTarget = NetUtils.createSocketAddr(mirrorNode) //下游节点的 P 地址 + 端口号
>+ mirrorSock = datanode.newSocket() //在本地创建一个 socket
>+ NetUtils.connect(mirrorSock, mirrorTarget, timeoutValue) //连接到下游节点
>+ OutputStream unbufMirrorOut = NetUtils.getOutputStream(mirrorSock, writeTimeout)
                                //socket 作为输出流
>+ InputStream unbufMirrorIn = NetUtils.getInputStream (mirrorSock)
                                // socket 也作为下游方向的输入流
>+ bos = new BufferedOutputStream(unbufMirrorOut, HdfsConstants.SMALL_BUFFER_SIZE)
                                //在 socket 输出流的基础上构筑带缓冲区的输出流
>+ mirrorOut = new DataOutputStream(bos) //再在此基础上构筑数据输出流
>+ mirrorIn = new DataInputStream(unbufMirrorIn) //在 socket 输入流基础上构筑数据输入流
>+ new Sender(mirrorOut).writeBlock(originalBlock, targetStorageTypes[0],
                                blockToken, clientname, targets, targetStorageTypes, srcDataNode, stage,
                                pipelineSize, minBytesRcvd, maxBytesRcvd, latestGenerationStamp,
                                requestedChecksum, cachingStrategy, false)
                                //创建一个 Sender,并调用其 writeBlock()方法
>+ mirrorOut.flush() //冲刷通向下游节点的输出流
>+ if (isClient) { //read connect ack (only for clients, not for replication req),若是由 App 发起
>+ connectAck = BlockOpResponseProto.parseFrom(PBHelper.vintPrefixed(mirrorIn))

```

```

//从下游方向读入 BlockOpResponseProto 响应报文
> ++ mirrorInStatus = connectAck.getStatus() //从响应报文中获取状态信息
> ++ firstBadLink = connectAck.getFirstBadLink() //获取流水线中的第一个失败点
> + }
> } //end if (targets.length > 0)
> if (isClient && !isTransfer) {
> + BlockOpResponseProto.newBuilder()...writeDelimitedTo(replyOut)
//构筑对用户的 BlockOpResponseProto 响应报文
> + replyOut.flush() //冲刷通向上游的应答输出流
> }
> //第三步,如果是数据块传输阶段,就接收、存储并转发数据内容
> if (blockReceiver != null) { //receive the block and mirror to the next target
> + String mirrorAddr = (mirrorSock == null)?null : mirrorNode
> + blockReceiver.receiveBlock(mirrorOut, mirrorIn, replyOut, mirrorAddr, ...)
> + if (isTransfer) writeResponse(SUCCESS, null, replyOut)
> }
> //否则,如果是 PIPELINE_CLOSE_RECOVERY,并且发起者是用户:
> if (isClient && stage == BlockConstructionStage.PIPELINE_CLOSE_RECOVERY) {
> + block.setGenerationStamp(latestGenerationStamp)
> + block.setNumBytes(minBytesRcvd)
> }
> //如果是 PIPELINE_CLOSE_RECOVERY,并且发起者是 DataNode
> // if this write is for a replication request or recovering a failed close for client, then
> // confirm block. For other client - writes, the block is finalized in the PacketResponder.
> if (isDatanode || stage == BlockConstructionStage.PIPELINE_CLOSE_RECOVERY) {
> + datanode.closeBlock(block, DataNode.EMPTY_DEL_HINT, storageUuid)
> }
> // 第四步,关闭所有的输入输出流
> IOUtils.closeStream(mirrorOut) //关闭下游方向的输出流
> IOUtils.closeStream(mirrorIn) //关闭下游方向的输出流
> IOUtils.closeStream(replyOut) //关闭上游方向的输出流
> IOUtils.closeSocket(mirrorSock) //关闭下游方向的 socket
> IOUtils.closeStream(blockReceiver) //关闭 blockReceiver
// (BlockReceiver 实现了 Closeable 界面)
> blockReceiver = null

```

这个操作方法的调用参数有 15 个之多,这些参数都是从上游发来的 OpWriteBlockProto 报文中提取出来的。同一项参数,在报文中的格式与在数据结构(对象)中的格式常常不同,所以这中间还要有格式转换。这是因为,其中有些参数是对象,这些对象得要被串行化了才能通过网络传输,而在到达目的地之后则要通过去串行化来还原。HDFS 为此提供了一个 PBHelper 类,里面有好多针对各种类的 convert() 函数,用来实施这些类的格式转换。不过我

们在这里不关心这样的细节。

OpWriteBlockProto 报文中最重要的成分是对数据块的描述,还原之后就是一个 ExtendedBlock 对象,其结构部分为:

```
class ExtendedBlock{
] String poolId           //数据块所属的 BlockPool,也就是所属的 NameNode
] Block block             //数据块的属性
]] long blockId           //块号
]] long numBytes          //块的当前长度
]] long generationStamp   //块的世代标记,类似于版本号
```

显然这里面并不包含数据块的内容,而只是一种描述。

其余参数的作用见摘要中的注释,其中有几个还需要稍加说明。

参数 clientname 是个字符串,如果此次 writeBlock()操作是由某个用户发起的,比方说是某个 App 要求写文件,那么这个 clientname 就是启动该 App 运行的用户名。但是有时候 DataNode 本身,出于复制数据块复份的目的,也可能会发起 writeBlock()操作,这样的操作不属于任何一个用户,此时 clientname 为空。所以,虽然操作的请求来自同一个节点,但是操作的发起者却可能不同。正因为这样,writeBlock()内部的局部量 isDatanode、isClient 都是从 clientname 来的。

参数 targets 和 targetStorageTypes 都是数组,前者是 DatanodeInfo 数组,后者是 StorageType 数组,两个数组大小相同,都取决于参数 pipelineSize。不难理解,这三个参数决定了流水线的长度和内容。除当前节点之外,数据块还要在几个节点上留下复份,要留几个复份, pipelineSize 就是几。数据块复份在流水线中不同的 DataNode 上可以有不同的存储类型,如磁盘、固态硬盘 SSD、RamDisk 等。另一个参数 storageType 则是该数据块在当前这个节点上的存储类型。

还有个参数 stage,是个 BlockConstructionStage 枚举类型对象,表示数据块在构筑过程中所处的阶段。

```
enum BlockConstructionStage {
] PIPELINE_SETUP_APPEND           // pipeline set up for block append
] PIPELINE_SETUP_APPEND_RECOVERY // pipeline set up for failed PIPELINE_SETUP_APPEND recovery
] DATA_STREAMING                 // data streaming
] PIPELINE_SETUP_STREAMING_RECOVERY // pipeline setup for failed data streaming recovery
] PIPELINE_CLOSE                  // close the block and pipeline
] PIPELINE_CLOSE_RECOVERY         // Recover a failed PIPELINE_CLOSE
] PIPELINE_SETUP_CREATE           // pipeline set up for block creation
] TRANSFER_RBW                    // transfer RBW for adding datanodes
] TRANSFER_FINALIZED              // transfer Finalized for adding datanodes
```

这些注释都是原有的。注意,前面三组是成对的,例如 PIPELINE_SETUP_APPEND 与

PIPELINE_SETUP_APPEND_RECOVERY,前者为偶数,后者为奇数,这两个枚举值就差最后一位。不过从 PIPELINE_SETUP_CREATE 开始的三个枚举值就不是那样了。

这所谓 stage,“阶段”,是什么意思呢?我们举个例来加以说明。一个 HDFS 文件当然可以有很多个数据块,除最后那个数据块之外,其他数据块的大小都是固定的,比方说 64MB,在第一块没有被写满 64MB 之前是不会开始写第二块的。但是最后那一块就不一样了,因为 App 在关闭文件之前未必会把最后一块写满,所以最后那一块往往是不满的。可是,虽然 HDFS 文件不支持随机写,却支持 append,过一段时间之后应用软件又可能打开这个文件并在其末尾添加数据。落实到 DataNode 的宿主文件系统,这就是打开一个数据块文件并在其末尾增添,直到达到 64MB 为止,那以后就要另起一个新数据块了。对于这样的 writeBlock() 操作,从数据块构筑的角度看,就是 PIPELINE_SETUP_APPEND。

另外,如前所述,一个 HDFS 文件的多个数据块一般会被分散存储在不同的节点上,而且每个数据块会有多个复份,以收“狡兔三窟”之效,为此在 writeBlock() 时要建立起一个流水线,即 pipeline,到完成了操作之后再予拆除。所谓 SETUP,就是指建立流水线的阶段。CLOSE 当然是关闭拆除流水线的阶段。而 DATA_STREAMING 和 TRANSFER,则是实际使用流水线的阶段。

这里的 RBW 也要作点说明,这是“Replica Being Written”的缩写。与此类似的还有:RWR 是“Replica Waiting to be Recovered”的缩写,RUR 是“Replica Under Recovery”经缩写。

大致明白了这些调用参数的意义和作用,我们就可以看代码的摘要了。我把 writeBlock() 操作的过程归纳成四步,并在摘要中做了注释,这里再补充做些说明。

第一步是根据需要创建 BlockReceiver 对象。为什么要创建 BlockReceiver 对象,什么情况下需要创建这个对象呢?我们在前面看到,DataXceiver 接收到的 OpWriteBlockProto 报文,里面只有关于数据和操作要求的描述,而并没有真正的数据,这说明后面必定还会有承载着数据的报文。显然我们需要有专门用来接收数据的程序,这些程序代码不同于我们已经看到的那些,因为所处理的问题和对象不同。本着模块化的原则,这里把相关的代码集中在一起定义成一个类,就是 BlockReceiver。但是这显然只是在有数据传输时才需要,所以在操作要求为例如 PIPELINE_CLOSE_RECOVERY 时是不需要的。不过也有例外,例如 PIPELINE_CLOSE_RECOVERY,如果操作的发起者是对方的 DataNode 本身而不是某个具体的用户,那也还是需要的,这一点我们在这里就不深究了。

第二步是根据需要建立与下游节点的连接并转发。如前所述,HDFS 文件系统中的数据都是“狡兔三窟”的,需要分存在不同的地方,所以在写数据时有个流水线,流水线中除最后一个节点外都负有接力转发的责任。所以,如果 targets.length 大于 0,就说明 targets 数组中还有别的节点,本节点不是最后一个节点,那就需要加以转发。正是在这种情景下,DataXceiver 需要对下游节点主动发起 WRITE_BLOCK 操作了,所以就得创建一个 Sender 对象并调用其 writeBlock() 操作方法。转发之后,还要等待来自下游节点的回答,即 BlockOpResponseProto 报文,从中获取有关连接状态的信息。其中有一项重要的信息是 FirstBadLink,就是下游流水线中的第一个失败的网络连接点,因为那一点之后的节点肯定都没有完成操作。如果操作的发起者为用户(而不是 DataNode),那还得把这些状态信息转发到上游方向,让 App 知道。

第三步是重点所在。如果在上面的第一步中创建了 BlockReceiver 对象,就通过其

receiveBlock()操作接收、存储并转发数据内容;否则就根据具体情况另做处理。具体的操作过程下面还要详述。注意,我们是在看下游节点上 writeBlock()的代码,其核心部分是通过另一个对象(模块)所提供的方法完成的,这个方法是 receiveBlock()。这就是说,下游节点上的 writeBlock()对于数据块的接收、存储、转发,其实是在 receiveBlock()中完成的。

第四步是善后,把操作过程中创建的输入输出流关闭掉,把 BlockReceiver 对象也关闭掉。BlockReceiver 类实现了 Closeable 界面,所以是“可关闭”的。注意这里还剩下了一个上游方向的输入流没有关闭,那是要由上游节点关闭的,因为是上游节点发起连接到本节点的,那就让它自己关闭。

下面就深入到 BlockReceiver。读者可能觉得 BlockReceiver 应该很简单,其实不然。我们知道典型 HDFS 数据块的大小是 64MB,但是网络上是以“包”即 packet 的方式传输的,一个 Block 得要分成许多个 Packet 来传输和整合。另外,且不说转发,至少得要把接收到的数据存储在本地的文件系统中,而存储的介质又分 DISK、SSD、RAMDISK 等类型,目录又有 tmp、rbw 和 finalized 之分。所以这个过程并不简单,而且对这个过程的理解对于深入理解 HDFS 的存储方案甚为重要。

我们先看 BlockReceiver 的数据结构部分:

```
class BlockReceiver implements Closeable {} //数据结构部分
[ ] DataInputStream in //上游方向的输入数据通道
[ ] DataChecksum clientChecksum //checksum used by client
[ ] DataChecksum diskChecksum //checksum we write to disk
[ ] OutputStream out //这是常规的输出流
[ ] FileDescriptor outFd
[ ] DataOutputStream checksumOut //这是 CRC 校验的输出流
[ ] PacketReceiver packetReceiver //Packet 接收模块
[ ] String inAddr, myAddr, mirrorAddr //相关节点的地址
[ ] DataOutputStream mirrorOut //下游方向的数据输出通道
[ ] Daemon responder //专门负责应答的线程
[ ] DataTransferThrottler throttle //数据传输速度(带宽)调节器
[ ] ReplicaOutputStreams streams //一对特殊的输出流,可以边写边进行 CRC 校验
[ ] OutputStream dataOut //用于数据的输出
[ ] OutputStream checksumOut //用于 CRC 校验的输出
[ ] DataChecksum checksum
[ ] ExtendedBlock block //the block to receive,正在接收的数据块描述
[ ] ReplicaInPipelineInterface replicaInfo //实现了 ReplicaInPipelineInterface 界面的某类对象
//事实上是个 ReplicaInPipeline 类对象
[ ] DataOutputStream replyOut //上游方向的输出数据通道
```

再看其操作部分:

```
class BlockReceiver implements Closeable {} //操作部分
[ ] BlockReceiver(final ExtendedBlock block, final StorageType storageType, ...)
[ ] verifyChunks(ByteBuffer dataBuf, ByteBuffer checksumBuf) //实施 CRC 检验
```



```

> clientChecksum.verifyChunkedSums(dataBuf, checksumBuf, clientname, 0)
] translateChunks(ByteBuffer dataBuf, ByteBuffer checksumBuf)
> diskChecksum.calculateChunkedSums(dataBuf, checksumBuf)
    //将用户的 CRC 检验 chunks 转换成磁盘上的 CRC 检验
] receivePacket() //接收一个 Packet
] manageWriterOsCache(long offsetInBlock) //对宿主系统文件读写缓冲的管理
] receiveBlock(DataOutputStream mirrOut, DataInputStream mirrIn, ...)
] class PacketResponder implements Runnable, Closeable {}
] class Packet {}
]] long seqno
]] boolean lastPacketInBlock
]] long offsetInBlock
]] long ackEnqueueNanoTime
]] Status ackStatus

```

对于 BlockReceiver 提供了一些什么操作,这个摘要可以给我们一个大致的印象。其中最重要的有前面在 writeBlock()中创建 BlockReceiver 对象时调用的构造函数 BlockReceiver()和 receiveBlock(),这二者都比较大,需要单独加以展开。另外,BlockReceiver 内部定义了专门负责应答的线程 PacketResponder,这个类也需要单独展开介绍,不适合在此就地展开。最后还要说一下 Packet,这是 BlockReceiver 内部定义的一个类,显然不同于网络层意义上的包 Packet,实际上这是作为 TCP 报文(Message)发送的,一个 BlockReceiver.Packet 在实际发送时可能需要分解成好多个网络层 Packet。

我们在前面看到,DataXceiver.writeBlock()的第一步操作是创建一个 BlockReceiver 对象,此时就会调用这个类的构造函数,这个构造过程又可分成几步:

```
[DataXceiver.run() > processOp() > writeBlock() > BlockReceiver.BlockReceiver()]
```

```

BlockReceiver.BlockReceiver(ExtendedBlock block, StorageType storageType,
    DataInputStream in, String inAddr, String myAddr,
    BlockConstructionStage stage, long newGs,
    long minBytesRcvd, long maxBytesRcvd,
    String clientname, DatanodeInfo srcDataNode, DataNode datanode,
    DataChecksum requestedChecksum, CachingStrategy cachingStrategy,
    boolean allowLazyPersist)
> this.isDatanode = clientname.length() == 0
> this.isClient = !this.isDatanode
> this.isTransfer = stage == BlockConstructionStage.TRANSFER_RBW
    || stage == BlockConstructionStage.TRANSFER_FINALIZED
> //第一步,根据不同情况创建文件或不同的 ReplicaInPipeline 对象
> if (isDatanode) { //replication or move,如果操作是由 DataNode 启动的
>+ replicaInfo = datanode.data.createTemporary(storageType, block)

```

```

== FsDatasetImpl.createTemporary(storageType, block)
    //在 tmp 目录下创建文件,并返回代表着这个 replica 的对象
    //返回值是个 ReplicaInPipeline 类对象,这是对抽象类 ReplicaInfo 的扩充
> } else { //操作由用户启动
>+ switch (stage) {
>+ case PIPELINE_SETUP_CREATE:
>++ replicaInfo = datanode.data.createRbw(storageType, block, allowLazyPersist)
    == FsDatasetImpl.createRbw(...) //在 rbw 目录下创建文件
    //返回值是个 ReplicaBeingWritten 类对象,这是对 ReplicaInPipeline 类的扩充
>++ datanode.notifyNamenodeReceivingBlock(block, replicaInfo.getStorageUuid())
    //向 NameNode 报告接收到数据块
>+ case PIPELINE_SETUP_STREAMING_RECOVERY:
>++ replicaInfo = datanode.data.recoverRbw(block, newGs, minBytesRcvd, maxBytesRcvd)
>++ block.setGenerationStamp(newGs) //设置数据块的世代标记
>+ case PIPELINE_SETUP_APPEND:
>++ replicaInfo = datanode.data.append(block, newGs, minBytesRcvd)
    == FsDatasetImpl.append(...) //在原有文件末尾添加
>++ if (datanode.blockScanner != null) { // remove from block scanner
>+++ datanode.blockScanner.deleteBlock(block.getBlockPoolId(), block.getLocalBlock())
    //从块扫描池中删去该数据块(因为未完成)
>++ }
>++ block.setGenerationStamp(newGs) //设置数据块的世代标记
>++ datanode.notifyNamenodeReceivingBlock(block, replicaInfo.getStorageUuid())
    //向 NameNode 报告接收到数据块
>+ case PIPELINE_SETUP_APPEND_RECOVERY:
>++ replicaInfo = datanode.data.recoverAppend(block, newGs, minBytesRcvd)
    == FsDatasetImpl.recoverAppend(...)
>++ if (datanode.blockScanner != null) { // remove from block scanner
>+++ datanode.blockScanner.deleteBlock(block.getBlockPoolId(), block.getLocalBlock())
    //从块扫描池中删去该数据块(因为未完成)
>++ }
>++ block.setGenerationStamp(newGs)
>++ datanode.notifyNamenodeReceivingBlock(block, replicaInfo.getStorageUuid())
>+ case TRANSFER_RBW, case TRANSFER_FINALIZED: //this is a transfer destination
>++ replicaInfo = datanode.data.createTemporary(storageType, block) //在 tmp 目录下创建文件
    == FsDatasetImpl.createTemporary(storageType, block)
>+ } //end switch
> } //end else
> //第二步,为目标数据块的复份存储打开或创建一组文件
> this.dropCacheBehindWrites = (cachingStrategy.getDropBehind() == null)?

```

```

        datanode.getDnConf().dropCacheBehindWrites :
        cachingStrategy.getDropBehind()
> this.syncBehindWrites = datanode.getDnConf().syncBehindWrites
> this.syncBehindWritesInBackground = datanode.getDnConf().syncBehindWritesInBackground
> isCreate = isDatanode || isTransfer ||
        stage == BlockConstructionStage.PIPELINE_SETUP_CREATE
        // isCreate 为 true 表示需新建文件,而不是对已有文件的 append
> streams = replicaInfo.createStreams(isCreate, requestedChecksum)
    == ReplicaInPipeline.createStreams(isCreate, requestedChecksum)
    //创建用来将此复份写入文件的输出流,这是个 ReplicaOutputStreams 对象
> //第三步,为 CRC 校验做好准备
> // read checksum meta information
> this.clientChecksum = requestedChecksum
> this.diskChecksum = streams.getChecksum() //这是 ReplicaOutputStreams.checksum
> this.needsChecksumTranslation = !clientChecksum.equals(diskChecksum) //不同就得转换
> this.bytesPerChecksum = diskChecksum.getBytesPerChecksum() //校验节(chunk)的大小
> this.checksumSize = diskChecksum.getChecksumSize() //校验数据的大小
> this.out = streams.getDataOut() //获取 ReplicaOutputStreams 中的数据输出流
> if (out instanceof FileOutputStream) { //这个数据输出流应该是个文件流
>+ this.outFd = ((FileOutputStream)out).getFD() //如果是文件流就记下其 FD
> } else { //如果不是文件流就错了
>+ LOG.warn("Could not get file descriptor for outputstream of class " + out.getClass())
> }
> bos = new BufferedOutputStream(streams.getChecksumOut(),
        HdfsConstants.SMALL_BUFFER_SIZE)
    //为 ReplicaOutputStreams 中的校验输出流构建一个缓冲输出流
> this.checksumOut = new DataOutputStream(bos)
    //以带缓冲的校验输出流作为 BlockReceiver 的校验输出流
> if (isCreate) { //write data chunk header if creating a new replica
>+ BlockMetadataHeader.writeHeader(checksumOut, diskChecksum) //写入元数据文件头部
        // The biggest part of data block metadata is CRC for the block.
> } //end if (isCreate)

```

DataXceiver 在接收到一个 WRITE_BLOCK 请求而调用其 writeBlock()操作时,虽然从总体上说都是数据块的写入,实际上却因数据块形成所处的阶段、原因和具体要求的不同而有不同的处理。

首先,如果上游节点中启动本次操作的实体是 DataNode 本身(而不是用户),那么其目的无非就是数据块的复制或转移,此时的复制或转移都是一对一、不涉及流水线操作的,全部目的就是在这个目标节点上存储一个数据块复份,传过来的是一个块的数据,存储在本地的宿主文件系统中就是一个数据文件和一个元数据文件,干干净净。所以这时候的操作就是通过 FsDatasetImpl.createTemporary()在 tmp 目录下创建文件。而若是由 App 启动的操作,那就

比较复杂了。

我们先看 `FsDatasetImpl.createTemporary()` 的摘要：

```
[DataXceiver.run() > processOp() > writeBlock() > BlockReceiver.BlockReceiver()
> FsDatasetImpl.createTemporary()]

FsDatasetImpl.createTemporary(StorageType storageType, ExtendedBlock b)
> ReplicaInfo replicaInfo = volumeMap.get(b.getBlockPoolId(), b.getBlockId())
> if (replicaInfo != null) {
>+ // 复份已经存在,但如果是 ReplicaInPipeline 且版本更老,那也是允许的
>+ if (replicaInfo.getGenerationStamp() < b.getGenerationStamp()
        && replicaInfo instanceof ReplicaInPipeline) {
>++ timeout = datanode.getDnConf().getXceiverStopTimeout()
>++ ((ReplicaInPipeline)replicaInfo).stopWriter(timeout) //Stop the previous writer
>++ invalidate(b.getBlockPoolId(), new Block[] {replicaInfo})
>+ } else { //否则就异常返回
>++ ReplicaAlreadyExistsException("Block " + b + " already exists in state " + ...) //异常返回
>+ }
> }
> FsVolumeImpl v = volumes.getNextVolume(storageType, b.getNumBytes())
    //根据所指定的存储类型和数据量从 volumes 中找到相应的文件卷
> // create a temporary file to hold block in the designated volume
> File f = v.createTmpFile(b.getBlockPoolId(), b.getLocalBlockId())
    //在此文件卷的 tmp 目录下创建临时文件
> ReplicaInPipeline newReplicaInfo = new ReplicaInPipeline(b.getBlockId(),
    b.getGenerationStamp(), v, f.getParentFile(), 0)
    //为此数据块复份创建一个 ReplicaInPipeline
> volumeMap.add(b.getBlockPoolId(), newReplicaInfo) //并将其加入 volumeMap 中
> return newReplicaInfo
```

注意,这里在创建临时文件的同时还创建了一个代表着这个复份的 `ReplicaInPipeline` 类对象 `newReplicaInfo`,而且函数的返回值就是这个 `newReplicaInfo`。`ReplicaInPipeline` 是对抽象类 `ReplicaInfo` 的扩充,所以 `newReplicaInfo` 同时也是个 `ReplicaInfo` 对象。这样,`BlockReceiver` 构造函数中的变量 `replicaInfo` 其实就是个 `ReplicaInPipeline`。

既然此时的全部目的就只是在本地留下作为数据块复份的文件,而不用进行流水线的接力操作,那怎么又要为其创建似乎是跟 Pipeline 有关的 `ReplicaInPipeline` 对象呢?进一步,返回到 `BlockReceiver` 的构造函数中以后,在下面的第二个步骤中要为此复份的写出创建一个输出流,这是理所当然的,但是为什么不是一般的文件输出流,而特别要由 `ReplicaInPipeline` 对象创建输出流呢?这 `ReplicaInPipeline` 是怎么个东西?

一个复份(Replica),在操作的过程中需要有个对象作为代表,光是 `Block` 中的那些信息还不够。按处理的目的、过程和状态的不同,数据块复份又有好几种,需要加以区分。为此

Hadoop 的代码中先定义了一个抽象类 `ReplicaInfo`,其数据结构部分的摘要如下:

```
class ReplicaInfo extends Block implements Replica {}
] FsVolumeSpi volume
] File baseDir
```

这样就把块与文件系统联系起来了。

然后又定义了对这抽象类的几种直接和间接的扩充:

```
class FinalizedReplica extends ReplicaInfo {}
class ReplicaInPipeline extends ReplicaInfo implements ReplicaInPipelineInterface {}
class ReplicaUnderRecovery extends ReplicaInfo {}
class ReplicaWaitingToBeRecovered extends ReplicaInfo {}
class ReplicaBeingWritten extends ReplicaInPipeline {}
```

显然,按设计者的意图,只要是与 Recovery 无关,又尚未 Finalized 的复份,就都属于 `ReplicaInPipeline`。而 `ReplicaBeingWritten`,则是一种特殊的 `ReplicaInPipeline`,那是真正需要通过流水线传到下游去的复份。

所以,尽管无须再往下游传输,这里所创建的也是个 `ReplicaInPipeline` 对象。事实上,无须再往下游传输复份的节点也可以是 Pipeline 中的最后一个节点,那就是 `InPipeline` 的一种特例。`ReplicaInPipeline` 的摘要如下:

```
class ReplicaInPipeline extends ReplicaInfo implements ReplicaInPipelineInterface {}
] long bytesAcked
] long bytesOnDisk
] byte[] lastChecksum
] Thread writer
] long bytesReserved
-----
] ReplicaInPipeline(Block block, FsVolumeSpi vol, File dir, Thread writer)
  > this( block.getBlockId(), block.getNumBytes(), block.getGenerationStamp(),
                                             vol, dir, writer, 0L)
>> super( blockId, len, genStamp, vol, dir)
>> this.bytesAcked = len
>> this.bytesOnDisk = len
>> this.writer = writer
>> this.bytesReserved = bytesToReserve
] setLastChecksumAndDataLen(long dataLength, byte[] lastChecksum)
] setWriter(Thread writer)
] createStreams(boolean isCreate, DataChecksum requestedChecksum)
```

那为什么 `ReplicaInPipeline` 需要提供自己的 `createStreams()` 方法来创建用于块文件的输出流,而不是采用一般的文件输出流呢?当然有其特殊之处,我们来看一下。

```
[DataXceiver.run() > processOp() > writeBlock() > BlockReceiver.BlockReceiver()
> ReplicaInPipeline.createStreams()]
```

```
ReplicaInPipeline.createStreams(boolean isCreate, DataChecksum requestedChecksum)
> File blockFile = getBlockFile() //注意,这个 File 是 java.io.File,基本上只是路径名
> File metaFile = getMetaFile()
> RandomAccessFile metaRAF = new RandomAccessFile(metaFile, "rw")
> if (!isCreate) { //For append or recovery, we must enforce the existing checksum.
>+ BlockMetadataHeader header = BlockMetadataHeader.readHeader(metaRAF)
>+ checksum = header.getChecksum()
>+ if (checksum.getBytesPerChecksum() != requestedChecksum.getBytesPerChecksum()) {
>++ IOException("Client requested checksum " + ...)
>+ }
>+ bytesPerChunk = checksum.getBytesPerChecksum()
>+ checksumSize = checksum.getChecksumSize()
>+ blockDiskSize = bytesOnDisk
>+ crcDiskSize = BlockMetadataHeader.getHeaderSize() +
                (blockDiskSize + bytesPerChunk - 1)/bytesPerChunk * checksumSize
>+ if (blockDiskSize > 0 &&
        (blockDiskSize > blockFile.length() || crcDiskSize > metaFile.length())) {
>++ IOException("Corrupted block: " + this)
>+ }
>+ checkedMeta = true
> } else {
>+ checksum = requestedChecksum //for create, we can use the requested checksum
> }
> bf = new RandomAccessFile(blockFile, "rw")
> blockOut = new FileOutputStream(bf.getFD())
> crcOut = new FileOutputStream(metaRAF.getFD())
> if (!isCreate) {
>+ blockOut.getChannel().position(blockDiskSize)
>+ crcOut.getChannel().position(crcDiskSize)
> }
> return new ReplicaOutputStreams(blockOut, crcOut,
                                checksum, getVolume().isTransientStorage())
    == ReplicaOutputStreams(OutputStream dataOut, OutputStream checksumOut,
                                DataChecksum checksum, boolean isTransientStorage)
>> this.dataOut = dataOut
>> this.checksumOut = checksumOut
>> this.checksum = checksum
```



```
>> this.isTransientStorage = isTransientStorage
```

原来,ReplicaInPipeline 要创建的输出流,实际上是绑定在一起的两个随机访问输出流,其中的块输出流 dataOut 就是普通的文件输出流,而另一个输出流 checksumOut 则是用于 CRC 校验的输出流。下面是 ReplicaOutputStreams 类的摘要:

```
class ReplicaOutputStreams implements Closeable {}
] OutputStream dataOut
] OutputStream checksumOut
] DataChecksum checksum
] boolean isTransientStorage
```

回到前面 BlockReceiver 的构造函数,那里为 ReplicaOutputStreams 的两个输出流添上缓冲,所形成的两个输出流 BlockReceiver.out 和 BlockReceiver.checksumOut 分别用于这个复份的数据文件和元数据文件。

再往上回到 DataXceiver.writeBlock(),对于无须接力转发的流程,就是由 DataNode 而不是 Client 发起的流程,就跳过其第二个步骤,下面就是对于 BlockReceiver.receiveBlock()的调用了。

```
[DataXceiver.run() > processOp() > writeBlock() > BlockReceiver.receiveBlock()]
```

```
BlockReceiver.receiveBlock(DataOutputStream mirrOut, DataInputStream mirrIn,
                           DataOutputStream replyOut, String mirrAddr,
                           DataTransferThrottler throttlerArg,
                           DatanodeInfo[] downstreams, boolean isReplaceBlock)
```

```
> syncOnClose = datanode.getDnConf().syncOnClose
> if (isClient &&!isTransfer) { //对于由 Client 发起的操作要有个应答线程
>+ responder = new Daemon(datanode.threadGroup,
                           new PacketResponder(replyOut, mirrIn, downstreams))
>+ responder.start(); // start thread to processes responses,启动应答线程 PacketResponder
> }
> while (receivePacket() >= 0) {} //Receive until the last packet,接收本块中的所有 Packet
    //然后
> if (responder != null) {
>+ ((PacketResponder)responder.getRunnable()).close() //关闭 PacketResponder
>+ responderClosed = true
> }
> if (isDatanode || isTransfer) {
>+ close() //close the block/crc files //关闭 CRC 校验文件
>+ block.setNumBytes(replicaInfo.getNumBytes())
>+ if (stage == BlockConstructionStage.TRANSFER_RBW) {
>++ datanode.data.convertTemporaryToRbw(block)
    //for TRANSFER_RBW, convert temporary to RBW,以供接力转发
```

```

>+ } else {
>+ datanode.data.finalizeBlock(block)
//for isDatnode or TRANSFER_FINALIZED, Finalize the block.
== FsDatasetImpl.finalizeBlock(ExtendedBlock b)
>+ }
> } //end while
> Thread.interrupted() //Clear the previous interrupt state of this thread.

```

这里的逻辑挺简单,就是在一个 while 循环中反复 receivePacket(), 一次一个 Packet, 直至完成整个块的接收(实际上是边接收边写盘);然后就视本次跨节点发送数据块复份的操作是由 Client 发起还是由 DataNode 发起,分别加以 convertTemporaryToRbw() 或者 finalizeBlock()。在我们现下的这个情景中,假定操作是由 DataNode 发起的增添或转移一个块的复份,所以就是 finalizeBlock()。

我们按次序来,先了解 receivePacket()。前面说过,这里的所谓 Packet, 与网络层的 Packet 是两码事,实际上相当于一个 Message,但是在 HDFS 中被视作一个基本的网络传输单元,实际发送时的进一步拆分那是 I/O 底层的事。

Hadoop 的源码中有个关于 Packet 格式的注释,稍作整理后是这样:

```

// PLEN      HLEN      HEADER      CHECKSUMS  DATA
// 32-bit 16-bit  <protobuf>  <variable length>
//
// PLEN:      Payload length = length(PLEN) + length(CHECKSUMS) + length(DATA)
//            This length includes its own encoded length in the sum for historical reasons.
// HLEN:      Header length = length(HEADER)
// HEADER:    the actual packet header fields, encoded in protobuf
// CHECKSUMS: the crcs for the data chunk. May be missing if checksums were not requested.
// DATA      the actual block data

```

最前面是 32 位的载荷长度;接着是 16 位的头长度;然后是由 ProtoBuf 生成的具体的 Header,里面有关于 checksum 和数据的信息;最后就是长度不定的校验码 checksum 和数据,checksum 在前,数据在后。

实际发送和接收的时候,HDFS 文件中的数据块都是定长的而且很大(默认为 64MB 或 128MB),只有最后那个块因未必写满而长度不定。所以一个块通常会被分成很多个 Packet 来发送和接收,BlockReceiver 管的是整个块的接收,而 Packet 的接收则有 PacketReceiver, BlockReceiver.receivePacket() 要依靠 PacketReceiver 才能完成。后者的摘要为:

```

class PacketReceiver implements Closeable {
    DirectBufferPool bufferPool
    ByteBuffer curPacketBuf //The entirety of the most recently read packet.
    ByteBuffer curChecksumSlice //A slice of curPacketBuf which contains just the checksums.
    ByteBuffer curDataSlice //A slice of curPacketBuf which contains just the data.
    PacketHeader curHeader
}

```

```

] receivePacket()
  > packetReceiver.receiveNextPacket(in)
  > ...
] receiveNextPacket(InputStream in) //接收下一个 Packet
  > doRead(null, in) == doRead(ReadableByteChannel ch, InputStream in)
] mirrorPacketTo(DataOutputStream mirrorOut) //将接收到的 Packet 通过另一个输出流写出
] reslicePacket(int headerLen, int checksumsLen, int dataLen) //切分 Packet 中的校验码和数据

```

在了解 BlockReceiver.receivePacket()对于 Packet 的接收和处理之前,我们预先了解一下 PacketReceiver 的 receiveNextPacket(),这是前者的基础,BlockReceiver.receivePacket()中首先就是调用这个函数。后面我们还会回过头来看 BlockReceiver.receivePacket()。

```

[DataXceiver.run() >processOp() >writeBlock() >BlockReceiver.receiveBlock()
>BlockReceiver.receivePacket()>PacketReceiver.receiveNextPacket()]

```

```

PacketReceiver.receiveNextPacket(InputStream in)
> doRead(null, in) == doRead(ReadableByteChannel ch, InputStream in)
>> curPacketBuf.clear() //清除缓冲区原有的状态
>> curPacketBuf.limit(PacketHeader.PKT_LENGTHS_LEN) //终点定在 PacketHeader 之前
>> doReadFully(ch, in, curPacketBuf) //第一次只是读入 PacketHeader 之前的部分
>>> if (ch != null) { //如果给定了 ReadableByteChannel,就从这个通道读入
>>>+ readChannelFully(ch, buf)
>>> } else { //否则就从输入流读入
>>>+ IOUtils.readFully(in, buf.array(), buf.arrayOffset() + buf.position(), buf.remaining())
>>>+ buf.position(buf.position() + buf.remaining())
>>> }
>> curPacketBuf.flip()//将缓冲区从写入翻转到读出状态
>> payloadLen = curPacketBuf.getInt() //读取载荷长度
>> dataPlusChecksumLen = payloadLen - Ints.BYTES
>> headerLen = curPacketBuf.getShort()
>> totalLen = payloadLen + headerLen
>> reallocPacketBuf(PacketHeader.PKT_LENGTHS_LEN +
                    dataPlusChecksumLen + headerLen) //根据实际需要分配缓冲区
>> curPacketBuf.clear()
>> curPacketBuf.position(PacketHeader.PKT_LENGTHS_LEN)
// 以 PacketHeader 的起点为起点
>> curPacketBuf.limit(PacketHeader.PKT_LENGTHS_LEN +
                    dataPlusChecksumLen + headerLen) //终点按 Packet 长度
>> doReadFully(ch, in, curPacketBuf) //第二次是读入 PacketHeader 以后的整个 Packet
>> curPacketBuf.flip()
>> curPacketBuf.position(PacketHeader.PKT_LENGTHS_LEN)

```

```

>>> byte[] headerBuf = new byte[headerLen]
>>> curPacketBuf.get(headerBuf)
>>> if (curHeader == null) curHeader = new PacketHeader()
>>> curHeader.setFieldsFromData(dataPlusChecksumLen, headerBuf)
>>> checksumLen = dataPlusChecksumLen - curHeader.getDataLen()
>>> reslicePacket(headerLen, checksumLen, curHeader.getDataLen())
>>>> lenThroughHeader = PacketHeader.PKT_LENGTHS_LEN + headerLen
>>>> lenThroughChecksums = lenThroughHeader + checksumsLen
>>>> lenThroughData = lenThroughChecksums + dataLen
>>>> curPacketBuf.position(lenThroughHeader) //Slice the checksums,提取校验码部分
>>>> curPacketBuf.limit(lenThroughChecksums)
>>>> curChecksumSlice = curPacketBuf.slice() == ByteBuffer.slice()
>>>> curPacketBuf.position(lenThroughChecksums) //Slice the data,提取数据部分
>>>> curPacketBuf.limit(lenThroughData)
>>>> curDataSlice = curPacketBuf.slice() == ByteBuffer.slice()
>>>> curPacketBuf.position(0) //将指针恢复到缓冲区的开头
>>>> curPacketBuf.limit(lenThroughData) //设置缓冲区的尽头

```

这样,当程序从 `PacketReceiver.receiveNextPacket()` 返回时,整个 `Packet` 已经读入缓冲区中,并已把校验码 `checksum` 和数据部分分别提取到 `curChecksumSlice` 和 `curDataSlice` 中。但是这二者还都只是在缓冲区中,还未加处理,既没有写到文件中,也没有转发。

在这个 `PacketReceiver.receiveNextPacket()` 的基础上,结合我们现在这个情景,再回头看 `BlockReceiver` 的 `receivePacket()`:

```

[DataXceiver.run() > processOp() > writeBlock() > BlockReceiver.receiveBlock()
> receivePacket()]

```

```
BlockReceiver.receivePacket()
```

```

> packetReceiver.receiveNextPacket(in) //read the next packet
  == PacketReceiver.receiveNextPacket(in) //见上,把一个 Packet 接收到了缓冲区中
> PacketHeader header = packetReceiver.getHeader()
> offsetInBlock = header.getOffsetInBlock()
> seqno = header.getSeqno()
> boolean lastPacketInBlock = header.isLastPacketInBlock() //是不是数据块中最后一个 Packet
> len = header.getDataLen()
> boolean syncBlock = header.getSyncBlock()
> firstByteInBlock = offsetInBlock //这个 Packet 所载的数据在数据块中的位置
> offsetInBlock += len
> if (replicaInfo.getNumBytes() < offsetInBlock) replicaInfo.setNumBytes(offsetInBlock)
> // put in queue for pending acks, unless sync was requested
> if (responder != null && !syncBlock && !shouldVerifyChecksum()) {

```

```

>+ ((PacketResponder) responder.getRunnable()).enqueue(segno,
                                astPacketInBlock, offsetInBlock, Status.SUCCESS)
                                //挂入应答线程 PacketResponder 的队列,供发送响应信息
> }
> //First write the packet to the mirror
> if (mirrorOut != null && !mirrorError) { //如果需要接力转发
>+ long begin = Time.monotonicNow()
>+ packetReceiver.mirrorPacketTo(mirrorOut) //将这个 Packet 按原样转发给下游节点
                                //包括数据和校验码两个部分
>+ mirrorOut.flush() //输出流 mirrorOut 创建于 DataXceiver.writeBlock()
> }
> ByteBuffer dataBuf = packetReceiver.getDataSlice() //这个 Packet 中的数据部分
> ByteBuffer checksumBuf = packetReceiver.getChecksumSlice() //CRC 校验码部分
> if (lastPacketInBlock || len == 0) { //如果是数据块中的最后一个 Packet
>+ LOG.debug("Receiving an empty packet or the end of the block " + block)
>+ if (syncBlock) flushOrSync(true)
> } else { //不是最后一个 Packet
>+ checksumLen = diskChecksum.getChecksumSize(len)
>+ checksumReceivedLen = checksumBuf.capacity()
>+ if (checksumReceivedLen > 0 && shouldVerifyChecksum()) { //有校验码,并要求校验
>++ verifyChunks(dataBuf, checksumBuf) //逐个 chunk 进行 CRC 验算
>++> clientChecksum.verifyChunkedSums(dataBuf, checksumBuf, clientname, 0)
                                == DataChecksum.verifyChunkedSums(ByteBuffer data, ByteBuffer checksums,
                                                                String fileName, long basePos)
>+ }
>+ if (needsChecksumTranslation) { //如果要求校验码转换(校验多项式不一样)
>++ translateChunks(dataBuf, checksumBuf)
                                //overwrite the checksums in the packet buffer with the appropriate
                                //polynomial for the disk storage
>+ }
> } //end if - else
> // by this point , the data in the buffer uses the disk checksum
> shouldNotWriteChecksum = checksumReceivedLen == 0 && streams.isTransientStorage()
> onDiskLen = replicaInfo.getBytesOnDisk()
> if (onDiskLen < offsetInBlock) {
>+ // finally write to the disk :
>+ if (onDiskLen % bytesPerChecksum != 0) { //prepare to overwrite last checksum
>++ adjustCrcFilePosition()
>+ }
>+ // If this is a partial chunk, then read in pre - existing checksum

```

```

>+ Checksum partialCrc = null
>+ if (!shouldNotWriteChecksum && firstByteInBlock % bytesPerChecksum != 0) {
    // If this is a partial chunk, then read in pre-existing checksum
>++ offsetInChecksum = BlockMetadataHeader.getHeaderSize() +
    onDiskLen / bytesPerChecksum * checksumSize
>++ partialCrc = computePartialChunkCrc(onDiskLen, offsetInChecksum)
>+ }
>+ startByteToDisk = (int)(onDiskLen - firstByteInBlock) +
    dataBuf.arrayOffset() + dataBuf.position()
>+ numBytesToDisk = (int)(offsetInBlock - onDiskLen)
>+ // Write data to disk.
>+ out.write(dataBuf.array(), startByteToDisk, numBytesToDisk) //写数据文件
>+ if (shouldNotWriteChecksum) { //如果这一次不应该写 CRC 校验文件
>++ lastCrc = null
>+ } else if (partialCrc != null) { //需要 CRC 校验文件
>++ lastCrc = null
>++ partialCrc.update(dataBuf.array(), startByteToDisk, numBytesToDisk)
>++ buf = FSOutputSummer.convertToByteArray(partialCrc, checksumSize)
>++ lastCrc = copyLastChunkChecksum(buf, checksumSize, buf.length) //留下这一次的 CRC
>++ checksumOut.write(buf) //写 CRC 校验文件,即元数据文件,扩展名为.meta
    //输出流 checksumOut 是在 BlockReceiver 的构造函数中创建的
>++ partialCrc = null
>+ } else {
>++ offset = checksumBuf.arrayOffset() + checksumBuf.position()
>++ end = offset + checksumLen
>++ lastCrc = copyLastChunkChecksum(checksumBuf.array(), checksumSize, end) //留下 CRC
>++ checksumOut.write(checksumBuf.array(), offset, checksumLen) //见上
>+ }
>+ }
> flushOrSync(syncBlock) //flush entire packet, sync if requested //冲刷输出流
> replicaInfo.setLastChecksumAndDataLen(offsetInBlock, lastCrc) //记录在 replicaInfo 中
> manageWriterOsCache(offsetInBlock)
> if (responder != null && (syncBlock || shouldVerifyChecksum())) { //要求应答线程发出响应
>+ ((PacketResponder) responder.getRunnable()).enqueue(seqno,
    lastPacketInBlock, offsetInBlock, Status.SUCCESS)
>+ }
> if (isReplaceBlock && (Time.monotonicNow() - lastResponseTime > responseInterval)) {
>+ response = BlockOpResponseProto.newBuilder().setStatus(Status.IN_PROGRESS)
>+ response.build().writeDelimitedTo(replyOut)
>+ replyOut.flush()

```



```

> }
> if (throttler != null) { // throttle I/O
>+ throttler.throttle(len)
> }
> return lastPacketInBlock? -1 : len //如果是最后一个 Packet 就返回 -1, 否则返回数据长度

```

可见, 每当程序从 `BlockReceiver.receivePacket()` 返回时, 都已经收到了一个 `Packet`, 对 `Packet` 中的数据和 CRC 校验码进行了验算, 并已写入了宿主文件系统中的相应块文件和元数据文件; 如果需要接力转发, 则也已经转发给 `Pipeline` 中的下一个节点。这样, 当 `BlockReceiver.receiveBlock()` 中的 `while` 循环结束时, 整个数据块的所有 `Packet` 都已得到处理, 剩下的只是关闭文件等善后的操作了。

在 `receivePacket()` 的过程中, 要有出错无非就是 I/O 方面的错, 例如网络断开连接、磁盘出问题等, 所以 `receivePacket()` 可能会有 `IOException` 异常, 而 `receivePacket()` 中确实也安排了对于 `IOException` 异常的处理, 不过我们在这里就不深入下去了。

结合 `BlockReceiver` 的 `receiveBlock()` 和 `receivePacket()`, 我们不妨也看一下 `BlockSender` 的 `sendBlock()` 和 `sendPacket()`。可想而知, 当我们想要把一个数据块的内容发送到另一个节点上时就要用到 `BlockSender.sendBlock()`, 这是对它的常规使用。但是这个函数还有个特殊的使用, 我们在前面 `DataNode` 那一章中看到, `DataNode` 在进行块扫描的时候也会调用 `BlockSender.sendBlock()`, 但那并非为了发送, 而只是为了对从磁盘上读出的数据进行 CRC 校验, 此时为 `BlockSender` 安排的输出流是个类似于 `dev/null` 这样的空输出流。

当然, 我们更关心的是它的常规使用, 就是节点间的数据块传输。

```

BlockSender.sendBlock(DataOutputStream out, OutputStream baseStream,
                      DataTransferThrottler throttler))

> initialOffset = offset
> OutputStream streamForSendChunks = out //一般是基于 Socket 的输出流, 也可以是空
> lastCacheDropOffset = initialOffset
> if (isLongRead() && blockInFd != null) { //读盘性能优化
>+ // Advise that this file descriptor will be accessed sequentially.
>+ NativeIO.POSIX.getCacheManipulator().posixFadviseIfPossible(
                      block.getBlockName(), blockInFd, ...)
> }
> manageOsCache() //Trigger readahead of beginning of file if configured.
    //上面这两个语句是针对宿主主机操作系统的, 有关底层文件数据缓存的优化
> pktBufSize = PacketHeader.PKT_MAX_HEADER_LEN
> boolean transferTo = transferToAllowed && !verifyChecksum
                      && baseStream instanceof SocketOutputStream
                      && blockIn instanceof FileInputStream
    //如果创建 BlockSender 时的参数 verifyChecksum 是 true, 则设置 transferTo 为 false
> if (transferTo) {
>+ FileChannel fileChannel = ((FileInputStream)blockIn).getChannel()

```

```

>+ blockInPosition = fileChannel.position()
>+ streamForSendChunks = baseStream
>+ maxChunksPerPacket = numberOfChunks(TRANSFER_TO_BUFFER_SIZE)
>+ pktBufSize += checksumSize * maxChunksPerPacket
           //Smaller packet size to only hold checksum when doing transferTo
> } else {
>+ maxChunksPerPacket = Math.max(
           1, numberOfChunks(HdfsConstants.IO_FILE_BUFFER_SIZE))
>+ pktBufSize += (chunkSize + checksumSize) * maxChunksPerPacket
           //Packet size includes both checksum and data
> }
> ByteBuffer pktBuf = ByteBuffer.allocate(pktBufSize) //分配一个足够大的 Packet 缓冲区
> while (endOffset > offset && !Thread.currentThread().isInterrupted()) {
           //把整个数据块(在输入流 blockIn 中)分成 Packet,一个个发送出去
>+ manageOsCache() //针对宿主操作系统,有关底层文件数据缓存的优化
>+ len = sendPacket(pktBuf, maxChunksPerPacket,
           streamForSendChunks, transferTo, throttler)
           == BlockSender.sendPacket(ByteBuffer pkt, int maxChunks, OutputStream out,
           boolean transferTo, DataTransferThrottler throttler)
           //在缓冲区 pkt 里形成一个 Packet,再从输出流 out 发送出去
           //内容来自输入流 blockIn 和 checksumIn,这是创建 BlockSender 对象时设置好的
>+> dataLen = (int) Math.min(endOffset - offset, (chunkSize * (long) maxChunks))
           //Packet 的长度取决于 chunkSize 和 maxChunks,除非已经没有那么多了
           //参数 maxChunks 就是上面的 maxChunksPerPacket
           //chunkSize 可以通过配置项“dfs.bytes-per-checksum”设置,一般是 512 字节
>+> numChunks = numberOfChunks(dataLen); // Number of chunks be sent in the packet
>+> checksumDataLen = numChunks * checksumSize
>+> packetLen = dataLen + checksumDataLen + 4
>+> boolean lastDataPacket = offset + dataLen == endOffset && dataLen > 0
>+> headerLen = writePacketHeader(pkt, dataLen, packetLen) //先在缓冲区 pkt 里写个头
>+> checksumOff = pkt.position() //checksum 在 pkt 中的起点(offset)
>+> byte[] buf = pkt.array()
>+> if (checksumSize > 0 && checksumIn != null) {
>+>+ readChecksum(buf, checksumOff, checksumDataLen) //从 checksumIn 读入校验码
>+>+ ...
>+> }
>+> dataOff = checksumOff + checksumDataLen //数据部分在 pkt 中的起点
>+> if (!transferTo) { //normal transfer
>+>+ IOUtils.readFully(blockIn, buf, dataOff, dataLen) //blockIn 是个 InputStream
           //从 blockIn 读入数据,这是在 BlockSender 的构造函数中创建的 FileInputStream

```

```

>+>+ if (verifyChecksum) {
>+>+ verifyChecksum(buf, dataOff, dataLen, numChunks, checksumOff)
    == BlockSender.verifyChecksum(byte[] buf, int dataOffset,
        int datalen, int numChunks, int checksumOffset)
    //若 CRC 校验出错会发起 ChecksumException
>+>+ }
>+> }
>+> if (transferTo) { //如果输出通道 out 是个 Socket,需要向下游转发
>+>+ SocketOutputStream sockOut = (SocketOutputStream)out
>+>+ sockOut.write(buf, headerOff, dataOff - headerOff)
>+>+ ...
>+> } else { //normal transfer,否则,out 是个文件输出流(甚至也可以是空输出流)
>+>+ out.write(buf, headerOff, dataOff + dataLen - headerOff)
    //在我们这个情景中,实际上是写到空输出通道,被丢弃了
>+> } //end of sendPacket()
>+ offset += len
>+ totalRead += len + (numberOfChunks(len) * checksumSize)
>+ seqno ++
> } //end while
> if (!Thread.currentThread().isInterrupted()) { //最后发一个空的 Packet,以示结束
>+ // send an empty packet to mark the end of the block
>+ sendPacket(pktBuf, maxChunksPerPacket, streamForSendChunks, transferTo, throttler)
>+ out.flush()
> }

```

从这里可以看到,一个数据块是分成好多 Packet 发送的,最后有个空的 Packet 标志着块的结束。但是怎样划分 Packet 呢? Packet 中数据的长度总是取 chunkSize 的整数倍(最后一个除外),或者说 Packet 中数据部分的边界一定也是 Chunk 的边界。Chunk 这个词,我们姑且把它翻译成“团”吧,因为“块”这个词已经被用掉了。之所以要有 Chunk,是为了 CRC 校验,因为一次 CRC 校验所覆盖的数据不宜太长。所以这里的思路是:最初是只有数据,没有校验码的,所以第一次写入文件或网络发送时在数据中每次向前做一个 Chunk 的 CRC 计算,得到一个校验码,就将它记载在元数据文件中;以后,每次从数据文件读出时同时也读元数据文件中的校验码,并逐个 Chunk 进行校验。如果要发送到别的节点,就在 Packet 中随同数据发送相关的 CRC 校验码。而 Chunk 的长度,太大了会降低 CRC 的效果,太小了又会开销太大,所以 Hadoop 在配置文件中有个配置项“dfs.bytes-per-checksum”,原始的设置是 512 字节。注意,CRC 校验码根本就不进入数据文件,对数据本身没有影响。而且,用或不用 CRC 校验,也是可以选择的。如前所述,DataNode 的块扫描操作也利用了 BlockSender.sendBlock(),但是把输出流设置成一个空流,相当于 dev/null,那就是只做校验不写输出。

```

[DataXceiver.run() > processOp() > writeBlock() > BlockReceiver.receiveBlock() >
BlockReceiver.finalizeBlock()]

```

```

BlockReceiver.finalizeBlock(long startTime)
> BlockReceiver.this.close()
>> packetReceiver.close()           //关闭 packetReceiver,断开本次连接
>> IOUtils.closeStream(checksumOut)   //关闭元数据文件
>> IOUtils.closeStream(out)           //关闭块文件
> long endTime = ClientTraceLog.isInfoEnabled()?System.nanoTime() : 0
> block.setNumBytes(replicaInfo.getNumBytes())
> datanode.data.finalizeBlock(block)
    == FsDatasetImpl.finalizeBlock(ExtendedBlock b)
> datanode.closeBlock(block, DataNode.EMPTY_DEL_HINT, replicaInfo.getStorageUuid())
>> BPOfferService bpos = blockPoolManager.get(block.getBlockPoolId())
>> bpos.notifyNamenodeReceivedBlock(block, delHint, storageUuid)
>> FsVolumeSpi volume = getFSDataset().getVolume(block)
>> if (blockScanner != null && !volume.isTransientStorage()) blockScanner.addBlock(block)

```

所以,BlockReceiver 这一层上的 finalize,首先当然是关闭块文件和元数据文件;然后是 FsDataset 这一层上的 finalize,主要是把添加的数据块复份记录到一些数据结构中去。最后是通知 NameNode,实际上是为下次向 NameNode 提交报告准备材料;并告知块扫描器这里又多了一个数据块。这里顺便也看一下 FsDatasetImpl.finalizeBlock()的摘要:

```

[DataXceiver.run() > processOp() > writeBlock() > BlockReceiver.receiveBlock() >
BlockReceiver.finalizeBlock() > FsDatasetImpl.finalizeBlock()]

```

```

FsDatasetImpl.finalizeBlock(ExtendedBlock b)
> replicaInfo = getReplicaInfo(b)
> if (replicaInfo.getState() == ReplicaState.FINALIZED) return
> finalizeReplica(b.getBlockPoolId(), replicaInfo)
>> if (replicaInfo.getState() == ReplicaState.RUR &&
    ((ReplicaUnderRecovery)replicaInfo).getOriginalReplica().getState()
    == ReplicaState.FINALIZED){
>>+ FinalizedReplica newReplicaInfo = (FinalizedReplica)
    ((ReplicaUnderRecovery)replicaInfo).getOriginalReplica()
>> } else {
>>+ FsVolumeImpl v = (FsVolumeImpl)replicaInfo.getVolume()
>>+ File f = replicaInfo.getBlockFile()
>>+ File dest = v.addFinalizedBlock(bpid, replicaInfo, f, replicaInfo.getBytesReserved())
    == FsVolumeImpl.addFinalizedBlock(bpid, replicaInfo, f, ...)
>>+ newReplicaInfo = new FinalizedReplica(replicaInfo, v, dest.getParentFile())
>>+ if (v.isTransientStorage()) {
>>++ ramDiskReplicaTracker.addReplica(bpid, replicaInfo.getBlockId(), v)

```

```

>>+ }
>> }
>> volumeMap.add(bpid, newReplicaInfo) == ReplicaMap.add(bpid, newReplicaInfo)
>> return newReplicaInfo

```

这就不用解释了。总之,接收到的数据块复份一经写入本地的宿主文件系统并加以封存(finalize),并进入了当地的有关数据结构,该复份在本节点的存储就算完成了。

以上所讲是基于由 DataNode(在 NameNode 的要求下)发起的数据块传输,目的是为其增添一个复份,此时只是单点对单点的传输,不采用流水线接力模式。

但是如果是由 Client 即 App 发起的数据块写入,那就要采取流水线模式了。这是因为,一个数据块需要有多个复份分别保存在不同的 DN 节点上,App 从 NM 得到的是一个 LocatedBlock 对象(数据结构),里面列举了需要存放的节点,而要求 App 逐一写入这些节点是不合理的,因为那样一来会对 App 的执行速度带来不利影响,二来也容易使网络中的流量趋于集中。所以,流水线式的接力传输和写入是比较合理的设计。

在接力传输一个来自 App 的数据块时,流水线中的各个 DN 节点首先要在其 PIPELINE_SETUP_CREATE 阶段执行一次 createRbw(),为这个数据块在宿主文件系统中建立一个 rbw 文件。所谓 rbw 文件,是创建在 rbw 子目录下的临时文件,rbw 意为 Replica Being Written:

```

[DataXceiver.run() > processOp() > writeBlock() > BlockReceiver.BlockReceiver()
> FsDatasetImpl.createRbw()]

```

```

FsDatasetImpl.createRbw(StorageType storageType,
                        ExtendedBlock b, boolean allowLazyPersist)
> ReplicaInfo replicaInfo = volumeMap.get(b.getBlockPoolId(), b.getBlockId())
> if (replicaInfo != null) { //同一数据块的复份业已存在
>+ ReplicaAlreadyExistsException("Block " + b + " already exists in state " + ...) //异常返回
> }
> // create a new block
> while (true) {
>+ if (allowLazyPersist) { //如果允许延迟持久化,就先写入 RamDisk
>++ // First try to place the block on a transient volume.
>++ FsVolumeImpl v = volumes.getNextTransientVolume(b.getNumBytes())
>++ datanode.getMetrics().incrRamDiskBlocksWrite()
>+ } else { //不允许延迟持久化,指定去哪就去哪
>++ FsVolumeImpl v = volumes.getNextVolume(storageType, b.getNumBytes())
>+ }
>+ break;
> } //end while
> // create an rbw file to hold block in the designated volume
> File f = v.createRbwFile(b.getBlockPoolId(), b.getLocalBlock()) //创建一个

```

```

    == FsVolumeImpl.createRbwFile(String bpid, Block b)
>> reserveSpaceForRbw(b.getNumBytes())
>> return getBlockPoolSlice(bpid).createRbwFile(b) == BlockPoolSlice.createRbwFile(Block b)
>>> f = new File(rbwDir, b.getBlockName()) //在 rbw 目录下
>>> return DatanodeUtil.createTmpFile(b, f) //创建一个临时文件
> ReplicaBeingWritten newReplicaInfo = new ReplicaBeingWritten(b.getBlockId(),
    b.getGenerationStamp(), v, f.getParentFile(), b.getNumBytes())
> volumeMap.add(b.getBlockPoolId(), newReplicaInfo)
> return newReplicaInfo

```

如果流水线是因为 PIPELINE_SETUP_APPEND 而建立,则是通过 FsDatasetImpl.append() 准备将接收到的数据添加到 rbw 文件的末尾:

```

[DataXceiver.run() > processOp() > writeBlock() > BlockReceiver.BlockReceiver()
> FsDatasetImpl.append()]

```

```

FsDatasetImpl.append(ExtendedBlock b, long newGS, long expectedBlockLen)
> if (newGS < b.getGenerationStamp()) { //GS 是世代标记(Generation Stamp)
>+ IOException("The new generation stamp " + newGS
    + " should be greater than the replica " + ...) //异常返回
> }
> ReplicaInfo replicaInfo = getReplicaInfo(b)
> if (replicaInfo.getState() != ReplicaState.FINALIZED) {
>+ ReplicaNotFoundException(ReplicaNotFoundException.UNFINALIZED_REPLICA + b)
> }
> if (replicaInfo.getNumBytes() != expectedBlockLen) {
>+ IOException("Corrupted replica " + replicaInfo + " with a length of " +
    replicaInfo.getNumBytes() + " expected length is " + expectedBlockLen)
> }
> return append(b.getBlockPoolId(), (FinalizedReplica)replicaInfo, newGS, b.getNumBytes())
>> // If the block is cached, start uncaching it.
>> cacheManager.uncacheBlock(bpid, replicaInfo.getBlockId())
>> // unlink the finalized replica
>> replicaInfo.unlinkBlock(1)
>> // construct a RBW replica with the new GS
>> File blkfile = replicaInfo.getBlockFile() //构筑最终的目标文件路径
>> FsVolumeImpl v = (FsVolumeImpl)replicaInfo.getVolume() //所属文件卷
>> if (v.getAvailable() < estimateBlockLen - replicaInfo.getNumBytes()) {
>>+ DiskOutOfSpaceException("Insufficient space for appending to " + replicaInfo)
>> }
>> File newBlkFile = new File(v.getRbwDir(bpid), replicaInfo.getBlockName())//临时文件名

```



```

>>> File oldmeta = replicaInfo.getMetaFile() //临时元数据文件名
>>> ReplicaBeingWritten newReplicaInfo = new ReplicaBeingWritten(replicaInfo.getBlockId(),
    replicaInfo.getNumBytes(), newGS, v, newBlkFile.getParentFile(),
    Thread.currentThread(), estimateBlockLen)
>>> File newmeta = newReplicaInfo.getMetaFile()
>>> // rename meta file to rbw directory
>>> NativeIO.renameTo(oldmeta, newmeta)
>>> // rename block file to rbw directory
>>> NativeIO.renameTo(blkfile, newBlkFile)
>>> // Replace finalized replica by a RBW replica in replicas map
>>> volumeMap.add(bpid, newReplicaInfo)
>>> v.reserveSpaceForRbw(estimateBlockLen - replicaInfo.getNumBytes())
>>> return newReplicaInfo

```

如前所述,BlockReceiver.BlockReceiver()在把数据块复份写入本地文件系统的同时还会向下游转发,直至到达 Pipe 中的最后一个节点。

14.2 命令 DNA_TRANSFER 的执行

我们在前一章中看到,DataNode 向 NameNode 发出心跳报告以后,NameNode 的响应中有可能搭载着命令 DNA_TRANSFER,要求这个 DataNode 将保存在当地的某个数据块复份发送给别的节点,在那里增添一个复份。而这个 DataNode 上与此 NameNode 相对应的 BPOfferService,则在 processCommandFromActive()中因此而调用 dn.transferBlocks(),即 DataNode.transferBlocks(),以完成跨节点的数据块复制。

```
[BPOfferService.processCommandFromActive() > DataNode.transferBlocks()]
```

```

DataNode.transferBlocks (String poolId, Block blocks[],
    DatanodeInfo xferTargets[[]], StorageType[[]]xferTargetStorageTypes)
> for (int i = 0; i < blocks.length; i++) { //对于 NameNode 交办的每个数据块
>+ transferBlock(new ExtendedBlock(poolId, blocks[i]),
    xferTargets[i], xferTargetStorageTypes[i])
>+> numTargets = xferTargets.length
>+> if (numTargets > 0) {
>+>+ xfersBuilder = new StringBuilder()
>+>+ for (int i = 0; i < numTargets; i++) { //将这个块的每个复制目的地都列入字符串数组
>+>+ xfersBuilder.append(xferTargets[i])
>+>+ xfersBuilder.append(" ")
>+>+ }
>+>+ LOG.info(bpReg + " Starting thread to transfer " + block + " to " + xfersBuilder)
>+>+ dtf = new DataTransfer(xferTargets, xferTargetStorageTypes, block,

```

```

lockConstructionStage.PIPELINE_SETUP_CREATE, "")
>+>+ new Daemon(dtf).start() //创建一个守护线程来执行 DataTransfer
>+> }
> }

```

对于需要加以复制的每个数据块, DataNode 都创建一个 DataTransfer 线程来完成此项工作。注意, 这里创建 DataTransfer 对象时的最后一个参数, 是个空字符串, 表示“委托人”是 DataNode 而不是某个 Client。

我们看 DataTransfer 类的摘要:

```

class DataNode.DataTransfer implements Runnable {}
] DatanodeInfo[] targets
] StorageType[] targetStorageTypes
] BlockConstructionStage stage
] String clientname
] CachingStrategy cachingStrategy
] DataTransfer(DatanodeInfo targets[], StorageType[] targetStorageTypes,
               ExtendedBlock b, BlockConstructionStage stage, String clientname)
] run()
  > String dnAddr = targets[0].getXferAddr(connectToDnViaHostname)
  > InetSocketAddress curTarget = NetUtils.createSocketAddr(dnAddr)
  > sock = newSocket()
  > NetUtils.connect(sock, curTarget, dnConf.socketTimeout)
  > Token<BlockTokenIdentifier> accessToken =
               BlockTokenSecretManager.DUMMY_TOKEN
  > OutputStream unbufOut = NetUtils.getOutputStream(sock, writeTimeout)
  > InputStream unbufIn = NetUtils.getInputStream(sock)
  > DataEncryptionKeyFactory keyFactory = getDataEncryptionKeyFactoryForBlock(b)
  > ... //建立加密层
  > out = new DataOutputStream(new BufferedOutputStream(unbufOut,
               HdfsConstants.SMALL_BUFFER_SIZE))
  > in = new DataInputStream(unbufIn)
  > blockSender = new BlockSender(b, 0, b.getNumBytes(), false, false, true,
               DataNode.this, null, cachingStrategy)
  > DatanodeInfo srcNode = new DatanodeInfo(bpReg)
  > new Sender(out).writeBlock(b, targetStorageTypes[0], accessToken, clientname,
               targets, targetStorageTypes, srcNode, stage, 0, 0, 0, 0,
               blockSender.getChecksum(), cachingStrategy, false)
  == Sender.writeBlock(ExtendedBlock blk, StorageType storageType,
               Token<BlockTokenIdentifier> blockToken, String clientName,
               DatanodeInfo[] targets, StorageType[] targetStorageTypes,

```

```

        DatanodeInfo source, BlockConstructionStage stage,
        int pipelineSize, long minBytesRcvd, long maxBytesRcvd,
        long latestGenerationStamp, DataChecksum requestedChecksum,
        CachingStrategy cachingStrategy, boolean allowLazyPersist)
    //注意,Sender.writeBlock()并非把一个Block发送出去
    //而只是发送一个WRITE_BLOCK控制报文
>>> header = DataTransferProtoUtil.buildClientHeader(blk, clientName, blockToken)
        //报头的构成,包括对数据块的描述等信息
>>> proto = OpWriteBlockProto.newBuilder() //构筑 OpWriteBlockProto 报文
        .setHeader(header) //首先是报头
        .setStorageType(PBHelper.convertStorageType(storageType))
        //然后是第一个节点的存储类型
        .addAllTargets(PBHelper.convert(targets, 1)) //流水线中的所有节点
        .addAllTargetStorageTypes(PBHelper.convertStorageTypes(targetStorageTypes, 1))
        //所有节点的存储类型
        .setStage(toProto(stage)) //本次数据块写入所处的阶段
        .setPipelineSize(pipelineSize) //流水线长度
        .setMinBytesRcvd(minBytesRcvd)
        .setMaxBytesRcvd(maxBytesRcvd)
        .setLatestGenerationStamp(latestGenerationStamp)
        .setRequestedChecksum(checksumProto)
        .setCachingStrategy(getCachingStrategy(cachingStrategy))
        .setAllowLazyPersist(allowLazyPersist) //是否延迟进行持久化
>>> proto.setSource(PBHelper.convertDatanodeInfo(source)) //产生数据的源节点
>>> msg = proto.build() //构建控制报文
>>> send(out, Op.WRITE_BLOCK, msg) //在控制报文前面添上操作码并发送
    //对方收到WRITE_BLOCK报文后会调用processOp(),然后DataXceiver.writeBlock()
> // send data & checksum
> blockSender.sendBlock(out, unbufOut, null) //然后才发送数据块
> // read ack
> if (isClient) { //如果是由Client发起则对方会有回应确认
>+ DNTransferAckProto closeAck =
        DNTransferAckProto.parseFrom(PBHelper.vintPrefixed(in))
>+ if (closeAck.getStatus() != Status.SUCCESS) {
>++ // 异常处理
>+ }
> }
> IOUtils.closeStream(blockSender)
> IOUtils.closeStream(out)
> IOUtils.closeStream(in)

```

```
> IOUtils.closeSocket(sock)
```

注意, `DataTransfer` 是个 `Runnable`, 前面程序中创建了一个 `Daemon` 线程来执行这个 `Runnable`, 使它实际上成为线程。这个线程有个 `run()` 函数, 但是这个 `run()` 函数里面并没有主循环, 它只是从头至尾做好一件事, 就是向一个目标发送一个数据块。具体的过程是先创建一个 `Socket`, 建立起与目标节点的连接, 再向其发送一个 `WRITE_BLOCK` 控制报文。然后才通过 `BlockSender.sendBlock()` 向对方发送这个数据块。

目标节点上的 `DataXceiver` 在接收到这个控制报文后会调用 `processOp()`, 然后因为操作码是 `WRITE_BLOCK` 就调用 `DataXceiver.writeBlock()`。而 `DataXceiver.writeBlock()` 则会创建一个 `BlockReceiver` 对象来与这一边对接, 这以后的过程我们在前面已经看到过了。

第15章

HDFS 的文件访问

15.1 DistributedFileSystem 和 DFSClient

关于 HDFS,本书前面讲的都是 NameNode、DataNode、FsImage、FSDirectory 这些,那是从 HDFS 内部如何构成如何运转的角度讲的。现在我们要讲一下它的外部表现,就是在使用者和应用层程序员眼里所看到的 HDFS。

Hadoop 的代码定义了一个类叫 DistributedFileSystem,这就是使用者和应用层程序员眼中所见的 HDFS。这个类是对抽象类 FileSystem 的扩充。之所以要先定义一个抽象类,然后再把它扩充成具体类 DistributedFileSystem,是因为 Hadoop 可以支持不止一种的文件系统,例如运行在 Amazon 的平台上时就有 S3FileSystem,对于宿主机的文件系统则有 RawLocalFileSystem,所有这些具体的文件系统都有相似的外观,例如都有 open()、append()、create()、rename()、delete()、setPermission()等这些功能和函数。这就好像 Linux 的 VFS 一样,本身并非一个具体的文件系统,但是却为所有的文件系统提供了一个轮廓。而 DistributedFileSystem,则是一种具体的文件系统,就是我们所说的 HDFS。与此类似,Hadoop 的代码中还定义了另一个抽象类 AbstractFileSystem,以及对此抽象类的扩充 Hdfs。Hdfs 与 DistributedFileSystem(以及 AbstractFileSystem 与 FileSystem)大同小异,例如 Hdfs 这个类就不提供 Snapshot。不过,这二者又有个共同点,就是 DistributedFileSystem 和 Hdfs 内部都有个 DFSClient 类的对象 dfs。DFSClient 是 HDFS 文件系统的用户即各种 App 与 NameNode 和 DataNode 交互的桥梁,针对 NameNode 和 DataNode 的 RPC 调用都是通过 DFSClient 类对象进行的。这意味着,Hadoop 集群内所有 App 对 HDFS 文件系统的操作最终都是通过 DFSClient 完成的。而 DistributedFileSystem 和 Hdfs 内部都有个 DFSClient 对象,就说明二者本可以实现完全相同的文件操作,之所以有所不同只是视角(View)的不同而已。注意,我们在说 HDFS 和 Hdfs 时所指的是不同的东西。

事实上,Hadoop 源码中大量使用的都是 DistributedFileSystem,实际上是 DistributedFileSystem(而不是 Hdfs)在各种 App 面前代表着 HDFS 文件系统。在 Hadoop 的代码中,常常可以看见对类型 FileSystem 的引用。之所以是引用抽象类 FileSystem 而不是具体类 DistributedFileSystem,是因为那也可以是扩充了抽象类 FileSystem 的别的具体类,例如也可以是 RawLocalFileSystem,或者也可以是 WebHdfsFileSystem。那么这个 FileSystem 对象究竟是哪一种具体类的对象呢?FileSystem 类提供了一个方法 get()供调用,所以 Hadoop 的代码中有很多对 FileSystem.get()的调用。

```

FileSystem.get(Configuration conf)
> uri = getDefaultUri(conf)
>> return URI.create(fixName(conf.get(FS_DEFAULT_NAME_KEY, DEFAULT_FS)))
> get(uri, conf) //另一个 FileSystem.get(), 多了一个参数 uri
>> String scheme = uri.getScheme() //例如 file://、http://、https://
>> String authority = uri.getAuthority()
>> if (scheme == null && authority == null) return get(conf) //use default FS
>> if (scheme != null && authority == null) { // no authority
>>+ URI defaultUri = getDefaultUri(conf)
>>+ if (scheme.equals(defaultUri.getScheme()) // if scheme matches default
    && defaultUri.getAuthority() != null) { // & default has authority
>>++ return get(defaultUri, conf)
>>+ }
>> }
>> String disableCacheName = String.format("fs. %s.impl.disable.cache", scheme)
>> if (conf.getBoolean(disableCacheName, false))
    return createFileSystem(uri, conf) //如果不启用缓存,就临时创建
>>> Class<?> clazz = getFileSystemClass(uri.getScheme(), conf)
>>>> if (!FILE_SYSTEMS_LOADED) loadFileSystems()
>>>> if (conf != null)
    clazz = (Class<?extends FileSystem>) conf.getClass("fs." + scheme + ".impl", null)
>>>> if (clazz == null) clazz = SERVICE_FILE_SYSTEMS.get(scheme)
>>>> if (clazz == null) throw new IOException("No FileSystem for scheme: " + scheme)
>>>> return clazz
>>> if (clazz == null) throw new IOException("No FileSystem for scheme: " + uri.getScheme())
>>> FileSystem fs = (FileSystem)ReflectionUtils.newInstance(clazz, conf) //创建具体类对象
>>> fs.initialize(uri, conf) == DistributedFileSystem.initialize(uri, conf)
    //通常是 DistributedFileSystem
>>>> super.initialize(uri, conf)
>>>> setConf(conf)
>>>> String host = uri.getHost()
>>>> homeDirPrefix = conf.get(DFSConfigKeys.DFS_USER_HOME_DIR_PREFIX_KEY,
    DFSConfigKeys.DFS_USER_HOME_DIR_PREFIX_DEFAULT)
    //分别是“dfs.user.home.dir.prefix”和“/user”
>>>> this.dfs = new DFSClient(uri, conf, statistics) //创建 DFSClient 类对象
>>>> this.uri = URI.create(uri.getScheme() + "://" + uri.getAuthority())
>>>> this.workingDir = getHomeDirectory()
>>> return fs
>> return CACHE.get(uri, conf) == Cache.get(uri, conf) //如启用了缓存,就从缓存获取
究竟采用何种文件系统是可以通过配置文件设置的,一般都是 DistributedFileSystem。

```


`FileSystem.get()`经常会得到调用,所以程序中设计了缓存,以免每次都要临时创建。但是缓存 `CACHE` 可以启用也可以不启用。如果启用,就通过 `Cache.get()`从缓存获取,这样就只需在首次调用这个函数时创建一次就行了。

一般而言,对 HDFS 文件的访问总是来自某种应用,不过应用也分两种:一种是用户提供的 App,包括 MapReduce 作业;另一种是由系统提供的工具,特别是它的各种 Shell 命令。事实上,以 MapReduce 作业为例,用户提供的 App 很少直接在程序中访问 HDFS 文件系统,而都是依靠 MapReduce 框架中所提供的某种 `InputFormat` 的 `RecordReader` 和 `OutputFormat` 的 `RecordWriter`;而用户提供的 Mapper 或 Reducer 则一般不会直接打开和访问 HDFS 文件。所以,这里我们用 HDFS 的 Shell,即 `FsShell` 作为实例来考察 HDFS 的文件访问。

15.2 FsShell

在 Hadoop 集群的节点上启动 `NameNode` 和 `DataNode` 之后,这二者完成了各自的初始化之后就静了下来,除心跳(`HeartBeat`)之外就没有多少活动了,因为这二者本质上都只是被动的服务提供者,不会主动“生事”。二者的区别只是一线服务者和二线服务者之分,由这二者构成的分布式文件系统 HDFS 从而也是被动的服务提供者。其实也不只是 HDFS,单机上的文件系统或网络中的文件服务器又何尝不是被动的服务者?

Hadoop 平台与单机操作系统在很大程度上是可以类比的。由 HDFS 提供的服务就像单机操作系统提供的文件系统服务;MapReduce 框架就像包括 C 语言程序库在内的运行环境;而具体的作业或应用,则类似于在单机操作系统上运行的应用软件。

这样类比一下就可发现,单机操作系统上还有个 Shell,它也是文件系统的使用者,为用户提供了许多使用和管理文件系统的手段;因而 Hadoop 按理也应有个 Shell。实际上 Hadoop 对于 Shell 的需求在某些方面部分地嫁接转移到了宿主机的 Shell 上。例如作业的启动,Hadoop 的作业提交机制其实起着一些类似于 Shell 的作用,这个过程的实质是通过单机的 Shell 启动了 Hadoop 的一部分类似于 Shell 的机制,它为 Hadoop 平台提交了 MapReduce 作业。

但是我们当然也需要有针对文件系统的 Shell 机制,使我们可以方便地进行一些针对 HDFS 的常规操作,例如在文件系统中创建目录、显示目录、删除文件、拷贝文件,特别是在 HDFS 与宿主机文件系统之间拷贝文件,等等。这一类的操作在单机操作系统上都是由一些称为 `utility` 的工具软件在 Shell 上完成的,Hadoop 显然也应该有相应的分布式 `utility` 和相应的 Shell,让用户能以命令行的方式启动执行这些工具软件。`FsShell` 就是这样一个 Shell,一个专门针对 HDFS 文件系统的 Shell。

我们看一下 `FsShell` 这个类的定义摘要:

```
class FsShell extends Configured implements Tool {
] FileSystem fs
] getFS()
    > fs = FileSystem.get(getConf()) //获取文件系统的类型
] main(String argv[])
    > FsShell shell = newShellInstance()
    >> return new FsShell()
```

```

> conf = new Configuration()
> res = ToolRunner.run(shell, argv) //ToolRunner.run()会回过来调用 FsShell.run()
] run(String argv[])
> init()
>> if (commandFactory == null) {
>>+ commandFactory = new CommandFactory(getConf())
>>+ commandFactory.addObject(new Help(), "- help")
>>+ commandFactory.addObject(new Usage(), "- usage")
>>+ registerCommands(commandFactory) == FsShell.registerCommands()
>>+> factory.registerCommands(FsCommand.class)
>>+>> m = registrarClass.getMethod("registerCommands", CommandFactory.class)
>>+>> m.invoke(null, this) == FsCommand.registerCommands(CommandFactory factory)
>>+>>> factory.registerCommands(AclCommands.class)
>>+>>> factory.registerCommands(CopyCommands.class)
>>+>>>> CopyCommands.registerCommands(CommandFactory factory)
>>+>>>>> factory.addClass(Merge.class, "- getmerge")
>>+>>>>> factory.addClass(Cp.class, "- cp")
>>+>>>>> factory.addClass(CopyFromLocal.class, "- copyFromLocal") //同 Put
>>+>>>>> factory.addClass(CopyToLocal.class, "- copyToLocal") //Get
>>+>>>>> factory.addClass(Get.class, "- get") //从 HDFS 拷贝一个文件到宿主文件系统
>>+>>>>> factory.addClass(Put.class, "- put") //从宿主文件系统拷贝一个文件到 HDFS
>>+>>>>> factory.addClass(AppendToFile.class, "- appendToFile")
>>+>>>> factory.registerCommands(Count.class)
>>+>>>> factory.registerCommands(Delete.class)
>>+>>>> factory.registerCommands(Display.class)
>>+>>>>> Display.registerCommands(CommandFactory factory)
>>+>>>>> factory.addClass(Cat.class, "- cat")
>>+>>>>> factory.addClass(Text.class, "- text")
>>+>>>>> factory.addClass(Checksum.class, "- checksum")
>>+>>>>> factory.registerCommands(FsShellPermissions.class)
>>+>>>>> factory.registerCommands(FsUsage.class)
>>+>>>>> factory.registerCommands(Ls.class)
>>+>>>>>> Ls.registerCommands(CommandFactory factory)
>>+>>>>>>> factory.addClass(Ls.class, "- ls")
>>+>>>>>>> factory.addClass(Lsr.class, "- lsr") // 整个子树的 ls, "r"表示递归
>>+>>>>>>> factory.registerCommands(Mkdir.class)
>>+>>>>>>> factory.registerCommands(MoveCommands.class)
>>+>>>>>>>> MoveCommands.registerCommands(CommandFactory factory)
>>+>>>>>>>>> factory.addClass(MoveFromLocal.class, "- moveFromLocal")
>>+>>>>>>>>> factory.addClass(MoveToLocal.class, "- moveToLocal")

```

```

>>+>>>>> factory.addClass(Rename.class, "-mv") //HDFS 内部的 mv
>>+>>>> factory.registerCommands(SetReplication.class) //设置文件存放份数
>>+>>>>> SetReplication.registerCommands(CommandFactory factory)
>>+>>>>> factory.addClass(SetReplication.class, "-setrep")
>>+>>>> factory.registerCommands(Stat.class)
>>+>>>> factory.registerCommands(Tail.class)
>>+>>>>> Tail.registerCommands(CommandFactory factory)
>>+>>>>> factory.addClass(Tail.class, "-tail")
>>+>>>> factory.registerCommands(Test.class)
>>+>>>> factory.registerCommands(Touch.class)
>>+>>>> factory.registerCommands(SnapshotCommands.class)
>>+>>>> factory.registerCommands(XAttrCommands.class)
>> }
> cmd = argv[0]
> Command instance = commandFactory.getInstance(cmd)
    == CommandFactory.getInstance(cmd)
> instance.run(Arrays.copyOfRange(argv, 1, argv.length)) //执行这条命令
    == Command.run(String...argv)
    //Command 是抽象类,实际执行的是具体命令的 run(String...argv)函数

```

注意,这里的 run() 内部并没有 while 循环,在 ToolRunner.run() 内部也没有这样的循环,所以这个 run() 内部的代码只被执行一次。也就是说,每启动一次 FsShell,其 run() 就被调用一次,执行完了就退出运行。下次再要运行 FsShell 命令就得再启动一次。

从代码中可见,FsShell 的每次运行都分两个阶段。第一个阶段是初始化阶段 init(),第二个阶段是调用为实现具体命令而定义的类的 run() 函数。初始化阶段的核心是 registerCommands(),通过 CommandFactory 登记了好多个种类的命令,例如 AclCommands、CopyCommands、Display、FsShellPermissions、Ls 等,每个种类以下又可以有多个具体的命令。例如 Ls 下面就有 -ls 和 -lsr,这二者是由 Ls 和 Lsr 两个类分别实现的,如果与 Linux 上的文件系统命令相比,则大致上相当于同一命令的不同选项。

这些命令的名称大多与 Unix/Linux 系统中相应 Shell 命令的名称相同。但是其中有几个需要特别加一点简单的说明:

copyFromLocal、put —— 从宿主机的文件系统拷贝一个文件到 HDFS。所谓 put,是站在宿主机的立场上而言,把本地的一个文件置入 HDFS。

copyToLocal、get —— 从 HDFS 拷贝一个文件到宿主文件系统。所谓 get,也是站在宿主机的立场上而言。

moveFromLocal、moveToLocal —— 与 copyFromLocal、copyToLocal 相似,但不是文件的拷贝而是文件的转移。

appendToFile —— 将宿主机文件系统中若干个文件的内容粘贴到一个 HDFS 文件末尾。

text —— 这是对 cat 的扩充,与 cat 相似但支持更多文件格式,包括 gzip 等压缩文件,还

有 Avro 和 TextRecord 格式。

setrep —— 设置 HDFS 文件中数据块的复份数量。如前所述, HDFS 是个分布的容错文件系统, 文件中的每个数据块都可以保存多份拷贝于不同的节点上, 但是具体保存几份则是可以设置的, 如果不作规定就默认 3 份。

显然, FsShell 需要有个类似于函数跳转表那样的数据结构, 以便根据命令名实现程序跳转。在面向对象的语言中, 这应该是一个对象(而不仅仅是数据结构)。这个对象, 就是 CommandFactory。读者不妨将其想象成一个结构数组, 数组的每个元素都是一个 KV 对: K 是命令名, 如“-cp”、“-cat”等; V 则是实现了这条命令的类, 如 class Cp、class Cat 等。不过 CommandFactory 的这个数组并非静态数组, 其内容需要在运行时动态登记填写, CommandFactory 提供了 registerCommands()、addObject()、addClass() 等方法用于命令登记。

所以, 在首次执行 FsShell.run() 之前需要通过 registerCommands() 登记所有的 FsShell 命令。FsShell 的命令分成很多类, 每个类包含一条或数条具体的命令, 例如 CopyCommands 类中有“-cp”等 7 条命令, Display 类中有“-cat”等 3 条命令, 但是 Tail 类中只有 1 条命令。这样, 每调用一次 registerCommands(), 就登记一个类的 FsShell 命令。所有实现这些命令的类, 都是对抽象类 Command 的直接或间接的扩充。

下面是抽象类 Command 类的摘要:

```
abstract class Command extends Configured {
    ] String[] args
    ] String name
    ] PrintStream out = System.out
    ] PrintStream err = System.err
    ] run(String...argv)
    ] processOptions(LinkedList<String> args)
    ] processRawArguments(LinkedList<String> args)
    ] expandArguments(LinkedList<String> args)
    ] processArguments(LinkedList<PathData> args)
    ] processPaths(PathData parent, PathData ... items)
```

向 CommandFactory 登记了所有的 FsShell 命令以后, 根据命令行中给定的命令名, 例如“-cat”, 通过 CommandFactory.getInstance() 就可以在 CommandFactory 中查到实现这条命令的 java 类, 并创建和返回一个该类的对象, 例如 Cat 类对象, 然后调用其 run() 函数, 就可以执行这条命令了。

CommandFactory.getInstance() 所返回的对象, 就是前面为具体命令所登记类型的对象, 那都是对 Command 类直接或间接的扩充, 所以对 Commomd.run() 的调用究竟是调用了为哪个类定义的 run() 函数, 要看具体的类是否定义了自己的 run() 函数, 或者从哪一个类继承了 run() 函数。进一步, Command 类的 run() 函数中会调用一些也是由 Command 类提供的函数, 例如 processArguments()、processPathArgument(), 是可以被为具体命令而定义的类所提供的同名函数覆盖的。所以, 即使 FsShell.run() 所调用的就是 Command 类中所定义的 run() 函数, 也仍有可能因为实现具体命令的类定义或继承了不同的 processArguments() 或者

processPathArgument()等函数而有不同的行为和表现。

以 CopyCommands 类内部定义的成分 Cp 和 Merge 为例:

Cp 类是对抽象类 CommandWithDestination 的扩充, CommandWithDestination 又是对抽象类 FsCommand 的扩充, FsCommand 则是对 Command 类的扩充, 这是 Cp 类的谱系:

```
abstract class Command extends Configured {}
abstract class FsCommand extends Command {}
abstract class CommandWithDestination extends FsCommand {}
class Cp extends CommandWithDestination
```

这样, 因为 Cp 并未提供 processArguments(), 但是 CommandWithDestination 提供了这个函数, 它就覆盖了 Command.processArguments()。于是在执行 Cp 命令的流程中所调用的 processArguments() 就是 CommandWithDestination.processArguments()。

这里涉及 FsCommand、CommandWithDestination 这些类的定义, 就不一一列举摘要了, 读者可以自己看一下。

再如 Merge, 则是对 FsCommand 的扩充。FsCommand 并未提供 processArguments(), 但是 Merge 本身提供了这个函数, 于是在执行 Merge 命令的流程中所调用的 processArguments() 就是 Merge.processArguments()。

FsShell 命令可以说是 HDFS 的典型使用者, 搞清楚其中几个命令的执行流程, 就大致上明白了 HDFS 的 API 设计与实现。我们先以命令 cat 为例考察其流程, Cat 类的谱系如下:

```
abstract class Command extends Configured {}
abstract class FsCommand extends Command {}
class Display extends FsCommand {}
class Cat extends Display {}
```

Cat 类并未提供自己的 run(String...argv) 函数, 而且 Display 和 FsCommand 均未提供这样的 run(String...argv) 函数, 所以 FsShell.run() 实际调用的是 Command.run(String...argv):

```
[FsShell.run() > Cat.run()]
```

```
Cat.run(String...argv) == Command.run(String...argv) //注意参数类型
> args = new LinkedList<String>(Arrays.asList(argv))
> processOptions(args) == Cat.processOptions(args) //由具体命令提供
>> ...
> processRawArguments(args)
>> e = expandArguments(args)
>> processArguments(e) //可以被具体命令的同名方法覆盖, 但是 Cat 并未覆盖
>>> for (PathData arg : args) {
>>>> processArgument(arg)
>>>> if (item.exists) processPathArgument(item) //可以被覆盖, 但是 Cat 并未覆盖
>>>>> processPaths(null, item)
>>>>>> for (PathData item : items) {
```

```

>>>+>>>+ processPath(item) == Cat.processPath(item) //Cat 覆盖了这个函数
>>>+>>>+> if (item.stat.isDirectory()) throw new PathIsDirectoryException(item.toString())
>>>+>>>+> item.fs.setVerifyChecksum(verifyChecksum)
>>>+>>>+> in = getInputStream(item) == Cat.getInputStream(PathData item)
>>>+>>>+>> return item.fs.open(item.path) == FileSystem.open(Path f)
>>>+>>>+>>> return open(f, getConf().getInt("io.file.buffer.size", 4096))//打开目标文件
== DistributedFileSystem.open(Path f, final int bufferSize)
//FileSystem 是抽象类,DistributedFileSystem 是其扩充和落实
//DistributedFileSystem 即 HDFS,打开文件后返回 FSDataInputStream
>>>+>>>+> printToStdout(in) //这是 Cat 的真正的功能所在
>>>+>>>+>> IOUtils.copyBytes(in, out, getConf(), false)
== IOUtils.copyBytes(InputStream in, OutputStream out,
int buffSize, boolean close)
>>>+>>>+>>> copyBytes(in, out, buffSize)
>>>+>>>+>>>> PrintStream ps = out instanceof PrintStream?(PrintStream)out : null
>>>+>>>+>>>> int bytesRead = in.read(buf) == FSDataInputStream.read(buf) //读文件
>>>+>>>+>>>> while (bytesRead >= 0) {
>>>+>>>+>>>>+ out.write(buf, 0, bytesRead) //写屏幕
>>>+>>>+>>>>+ bytesRead = in.read(buf) == FSDataInputStream.read(buf) //读文件
>>>+>>>+>>>> }
>>>+>>>+ if (recursive && item.stat.isDirectory()) recursePath(item)
>>>+>>>+> processPaths(item, item.getDirectoryContents()) //递归
>>>+>>>+ postProcessPath(item)
>>>+>>> } //end for (PathData item : items)
>>>+> else processNonexistentPath(item)
>>> } //end for (PathData arg : args)

```

我们在这里不关心对于命令行选项和目标路径的处理,也不关心当一个目标是目录的时候是否递归显示这个目录中的文件。我们关心的是,Cat 这个命令最终会打开 HDFS 文件系统中的某个文件,并从文件中读出。更确切地说,是打开 HDFS 文件后返回一个 FSDataInputStream,然后从这个 FSDataInputStream 中读出,这就是在访问 HDFS 文件系统。

别的 App,包括用户提供的 App,无论直接还是间接,最终都是这样打开和读 HDFS 文件的。

15.3 HDFS 的打开文件流程

就如我们在上面 FsShell 命令 Cat 的代码摘要中所见,当 App 要打开一个 HDFS 文件时,就调用 fs.open(),这个 fs 就是 org.apache.hadoop.fs,就是 DistributedFileSystem,所以实际调用的是 DistributedFileSystem.open():

DistributedFileSystem.open(Path f, int bufferSize)

```
> Path absF = fixRelativePart(f)
> Resolver = new FileSystemLinkResolver<FSDataInputStream>() //创建此类对象
] doCall(final Path p)
    > DFSInputStream dfsis = dfs.open(getPathName(p), bufferSize, verifyChecksum)
    > return dfs.createWrappedInputStream(dfsis)
        //返回一个 FSDataInputStream
] next(FileSystem fs, Path p)
    > return fs.open(p, bufferSize)
> Resolver.resolve(this, absF) == FileSystemLinkResolver.resolve(FileSystem filesystems, Path path)
>> Path p = path
>> FileSystem fs = filesystems
>> for (boolean isLink = true; isLink;) {
>>+ try {
>>++ in = doCall(p)
>>++> DFSInputStream dfsis = dfs.open(getPathName(p), bufferSize, verifyChecksum)
        //dfs 是在 DistributedFileSystem 初始化时创建的 DFSCClient 类对象
        == DFSCClient.open(String src, int buffersize, boolean verifyChecksum)
        //在此过程中可能会有多种异常,下面捕捉的是因符号连接解析失败而致的异常
>>++> return dfs.createWrappedInputStream(dfsis)
>>++ isLink = false //如果 doCall()成功,就不继续循环了
>>+} catch(UnresolvedLinkException e){ //如果是因为符号连接解析失败
>>++ p = FSLinkResolver.qualifySymlinkTarget(fs.getUri(), p, filesystems.resolveLink(p))
        //把文件路径换成符号连接的目标
>>++ fs = FileSystem.getFSofPath(p, filesystems.getConf()) //获知该路径所在的文件系统
>>++ if (!fs.equals(filesystems)) return next(fs, p) //如果不是在同一文件系统中
>>++> return fs.open(p, bufferSize) //就调用那一种文件系统的 open()函数
>>+ } //如果是在同一文件系统中,就继续循环
>> } //end for
>> return in
```

这段代码摘要的意思是:创建一个 `FileSystemLinkResolver` 类的对象,然后调用其 `resolve()` 函数。这个 `FileSystemLinkResolver` 类是个抽象类,所以需要动态补上两个函数的定义,那就是 `doCall()` 和 `next()`。这里的操作集中在对于文件系统中符号连接的解析,这个解析的过程,即 `doCall()`,放在一个 `try{}catch{}结构` 中;一般我们在做摘要的时候不太注意异常,但是这里的情况有点特殊,实际上是把异常捕捉机制用作了 `if-else`。

真正的打开文件操作,是通过 `DFSCClient.open()` 完成的。对于 HDFS, `DFSCClient` 是其 Client;对于 App,则 `DFSCClient` 相当于访问 HDFS 的桥梁。HDFS 的使用者与 `NameNode` 的交互和对 `DataNode` 的数据读/写均需通过 `DFSCClient` 进行。所以我们接着往下看 `DFSCClient.open()` :

```
[DistributedFileSystem.open() > DFSClient.open()]
```

```
DFSClient.open(String src, int buffersize, boolean verifyChecksum)
> checkOpen() //检查文件系统是否在运行
> return new DFSInputStream(this, src, buffersize, verifyChecksum)
    == DFSInputStream(DFSClient dfsClient, String src, int buffersize, boolean verifyChecksum)
    // DFSInputStream 对象的构造方法
>> this.dfsClient = dfsClient
>> this.verifyChecksum = verifyChecksum
>> this.buffersize = buffersize
>> this.src = src
>> this.cachingStrategy = dfsClient.getDefaultReadCachingStrategy()
>> openInfo()
>>> lastBlockBeingWrittenLength = fetchLocatedBlocksAndGetLastBlockLength()
>>>> LocatedBlocks newInfo = dfsClient.getLocatedBlocks(src, 0) //从 offset = 0 开始
>>>>> getLocatedBlocks(src, start, dfsClientConf.prefetchSize)
>>>>>> callGetBlockLocations(namenode, src, start, length)
>>>>>>> return namenode.getBlockLocations(src, start, length) //对 NameNode 的 RPC 调用
>>>> if (newInfo == null) throw new IOException("Cannot open filename " + src) //失败
>>>> if (locatedBlocks != null) { //如果以前就有这个信息在本地,就新老比较:
>>>>+ Iterator<LocatedBlock> oldIter = locatedBlocks.getLocatedBlocks().iterator()
>>>>+ Iterator<LocatedBlock> newIter = newInfo.getLocatedBlocks().iterator()
>>>>+ while (oldIter.hasNext() && newIter.hasNext()) {
>>>>++ if (!oldIter.next().getBlock().equals(newIter.next().getBlock())) {
>>>>+++ throw new IOException("Blocklist for " + src + " has changed!")
>>>>++ }
>>>>+ }
>>>> }
>>>> locatedBlocks = newInfo
>>>> ...
>>>>> fileEncryptionInfo = locatedBlocks.getFileEncryptionInfo()
>>>>> return lastBlockBeingWrittenLength
- - - - - 返回到了 openInfo(), 返回值在 lastBlockBeingWrittenLength 中 - - - - -
>>> retriesForLastBlockLength = dfsClient.getConf().retryTimesForGetLastBlockLength
>>> while (retriesForLastBlockLength > 0) {
>>>+ if (lastBlockBeingWrittenLength == -1) { //如果上一次的尝试失败
>>>++ waitFor(dfsClient.getConf().retryIntervalForGetLastBlockLength) //过会儿再试
>>>++ lastBlockBeingWrittenLength = fetchLocatedBlocksAndGetLastBlockLength()
>>>+ } else break //如果 lastBlockBeingWrittenLength >= 0 就跳出循环
>>> retriesForLastBlockLength --
```

```

>>>> } //end while
>>>> if (retriesForLastBlockLength == 0) {
>>>>+ throw new IOException("Could not obtain the last block locations.") //打开文件失败
>>>> }

```

摘要中加了注释,读者理解起来应该不会有困难。从中可以看出,对于 HDFS 文件的 open 操作,其实只是通过 RPC 向 NameNode 索要目标文件各数据块的存储地点和文件长度。所谓 LocatedBlock,意思就是已经定位的块,而 LocatedBlocks 是 LocatedBlock 的复数。

LocatedBlock 数据结构部分的摘要是这样:

```

class LocatedBlock{
] ExtendedBlock b
] long offset; //offset of the first byte of the block in the file,在文件中的位置
] DatanodeInfo[] locs //这个数据块的各个复份存放的地点
] String[] storageIDs //这个数据块的各个复份所在存储设备的 ID
] StorageType[] storageTypes //这个数据块的各个复份的存储类型,如 RAMDISK
] DatanodeInfo[] cachedLocs //List of cached datanode locations,在哪几个节点上有缓存

```

随着对 NameNode 的 RPC 调用,现在“球”滚到了 NameNode 这一边。我们略去 PB 层和 RPC 层的操作,直接看 NameNode 上的 getBlockLocations():

请求报文的到来使 NameNode 这一边 PB 层的 getBlockLocations()受到调用:

```

[DFSClient.open> openInfo()>DFSInputStream.fetchLocatedBlocksAndGetLastBlockLength()
=> NameNodeRpcServer.getBlockLocations()]

NameNodeRpcServer.getBlockLocations(src, offset, length)
> return namesystem.getBlockLocations(getClientMachine(), src, offset, length)
== FSNamesystem.getBlockLocations(String clientMachine,
String src, long offset, long length) //offset 为 0
>>> LocatedBlocks blocks = getBlockLocations(src, offset, length, true, true, true)
>>>> return getBlockLocationsInt(src, offset, length,
doAccessTime, needBlockToken, checkSafeMode)
>>>>> LocatedBlocks ret =
getBlockLocationsUpdateTimes(src, offset, length, doAccessTime, needBlockToken)
//主要靠此操作完成
>>> if (blocks != null) {
>>>+ blockManager.getDatanodeManager().sortLocatedBlocks(
clientMachine, blocks.getLocatedBlocks())
//按离 clientMachine 节点的远近排序
>>>+ // lastBlock is not part of getLocatedBlocks(), might need to sort it too
>>>+ LocatedBlock lastBlock = blocks.getLastLocatedBlock()
>>>+ if (lastBlock != null) {
>>>++ ArrayList<LocatedBlock> lastBlockList = Lists.newArrayListWithCapacity(1)

```

```

>> ++ lastBlockList.add(lastBlock)
>> ++ blockManager.getDatanodeManager().sortLocatedBlocks(clientMachine, lastBlockList)
>> + }
>> }
>> return blocks

```

显然,操作的核心在于 `getBlockLocationsUpdateTimes()`:

```

[NameNodeRpcServer.getBlockLocations() > FSNamesystem.getBlockLocations()
> getBlockLocationsInt() > getBlockLocationsUpdateTimes()]

```

```

FSNamesystem.getBlockLocationsUpdateTimes(String srcArg, long offset, long length,
                                           boolean doAccessTime, boolean needBlockToken)

```

```

> String src = srcArg
> FSPermissionChecker pc = getPermissionChecker() //用来检查访问权限
> byte[][] pathComponents = FSDirectory.getPathComponentsForReservedPath(src)
> for (int attempt = 0; attempt < 2; attempt++) {
> + boolean isReadOp = (attempt == 0)
> + if (isReadOp) { // first attempt is with readlock
> ++ checkOperation(OperationCategory.READ)
> ++ readLock()
> + } else { // second attempt is with write lock
> ++ checkOperation(OperationCategory.WRITE)
> ++ writeLock(); // writelock is needed to set accesstime
> + }
> + src = resolvePath(src, pathComponents)
> + if (isReadOp) checkOperation(OperationCategory.READ) //访问权限检验
> + else checkOperation(OperationCategory.WRITE)
> + if (isPermissionEnabled) checkPathAccess(pc, src, FsAction.READ) //访问权限检验
> + // if the namenode is in safemode, then do not update access time
> + if (isInSafeMode()) doAccessTime = false
> + INodesInPath iip = dir.getInNodesInPath(src, true)
> + INode[] inodes = iip.getInNodes()
> + INodeFile inode = INodeFile.valueOf(inodes[inodes.length-1], src) //目标文件的 INode
> + if (isPermissionEnabled) checkUnreadableBySuperuser(pc, inode, iip.getPathSnapshotId())
> + if (!iip.isSnapshot() //snapshots are readonly, so don't update atime.
                                && doAccessTime && isAccessTimeSupported()) {
> ++ long now = now()
> ++ if (now > inode.getAccessTime() + getAccessTimePrecision()) {
> +++ if (isReadOp) continue
> +++ boolean changed = dir.setTimes(inode, -1, now, false, iip.getLatestSnapshotId())

```

```

>+++ if (changed) getEditLog().logTimes(src, -1, now)
>+++ }
>+ }
>+ long fileSize = iip.isSnapshot()?inode.computeFileSize(iip.getPathSnapshotId())
        : inode.computeFileSizeNotIncludingLastUcBlock()
>+ boolean isUc = inode.isUnderConstruction()
>+ if (iip.isSnapshot()) {
>++ length = Math.min(length, fileSize - offset)
>++ isUc = false
>+ }
>+ FileEncryptionInfo feInfo = FSDirectory.isReservedRawName(srcArg)?null
        : dir.getFileEncryptionInfo(inode, iip.getPathSnapshotId(), iip)
>+ BlockInfo[] blks = inode.getBlockInfo() //从目标文件的 INode 对象中获取其 BlockInfo[]
>+ LocatedBlocks blocks = blockManager.createLocatedBlocks(blks, fileSize,
        isUc, offset, length, needBlockToken, iip.isSnapshot(), feInfo)
        //根据 BlockInfo[] 创建 LocatedBlocks, 其核心为 List<LocatedBlock>
>+ // Set caching information for the located blocks.
>+ for (LocatedBlock lb: blocks.getLocatedBlocks()) {
>++ cacheManager.setCachedLocations(lb)
>+ }
>+ return blocks
> } //end for
> return null; // can never reach here

```

这里的访问权限控制等当然很重要,但却并非我们此刻所关心。对于我们现在这个情景而言,重要的是按目标文件路径在目录中找到其 INodeFile,从而获得这个文件的 BlockInfo 数组,这就是整个操作的核心所在,LocatedBlocks 对象就是在此基础上创建的。

最后,这个 LocatedBlocks 对象被逐层返回到 PB 层的 getBlockLocations(),在那里被转换成响应报文;再将响应报文返回给 RPC 层,发送回 Client 一方。

可见,对于 NameNode 而言,所谓“打开文件”这个操作只是获取一个文件的 BlockInfo 数组而已,在 NameNode 上甚至都没有留下什么痕迹。如果此后对这个文件没有什么改变的话,也就无所谓关闭不关闭。不过,如果要对这文件进行写操作,那就不一样了。

15.4 HDFS 的读文件流程

“球”又回到 Client 一方。由于响应报文的到来,正在睡眠等待的线程 Cat 被唤醒,经由 RPC 层和 PB 层的处理,从响应报文中恢复出所承载的数据结构,然后返回到 DFSInputStream 的 fetchLocatedBlocksAndGetLastBlockLength() 中,所返回的就是 NameNode 那里临时创建的 LocatedBlocks 对象 newInfo,这个对象被赋值给 DFSInputStream 内部的结构成分 locatedBlocks。如前所述,LocatedBlocks 中对于目标文件的每个数据块都列出了它每个备份

所在的节点、存储类型、是否有缓存等信息。

从 NameNode 获取这些信息之后,就 Cat 命令而言,下一步就是要依次从其中所含的每个数据块读取数据。这些块都有多个复份,存放在不同的节点上。不过 API 层面上的使用者无须关心这些,他们只需从已打开的 InputStream 中读取就行了,下层的操作自有 InputStream 承担。具体到 Cat 命令,是依次从 InputStream 中读,往 OutputStream 里写,这是一种比较具有典型性的操作,所以工具类 IOUtils 专门为此提供了一种操作方法 copyBytes()。

```
[Cat.run() > processRawArguments() > processArguments() > processArgument()
> processPathArgument() > processPaths() > printToStdout() > IOUtils.copyBytes()]
```

```
IOUtils.copyBytes(InputStream in, OutputStream out, long count, boolean close)
> byte buf[] = new byte[4096]
> bytesRemaining = count
> while (bytesRemaining > 0) {
>+ bytesToRead = (int)(bytesRemaining < buf.length?bytesRemaining : buf.length)
>+ bytesRead = in.read(buf, 0, bytesToRead) //从输入流 DFSInputStream 读入
== DFSInputStream.read(byte buf[], int off, int len)
>+ if (bytesRead == -1) break //读文件失败,结束循环
>+ out.write(buf, 0, bytesRead) //写入输出流,那就是 stdout
>+ bytesRemaining -= bytesRead
> }
> if (close) {
>+ out.close()
>+ in.close()
> }
```

在执行 Cat 命令的情景中,输入流 in 是前面通过 DFSCClient.open() 获得的 DFSInputStream, 代表着已打开的目标文件;输出流 out 则就是宿主操作系统上的标准输出通道 stdout, 实际上就是控制台显示屏。

所以,执行 Cat 命令时的文件内容是通过 DFSInputStream.read() 读入的:

```
[IOUtils.copyBytes() > DFSInputStream.read()]
```

```
DFSInputStream.read(byte buf[], int off, int len)
> byteArrayReader = new ByteArrayStrategy(buf)
> return readWithStrategy(byteArrayReader, off, len)
== DFSInputStream.readWithStrategy(ReaderStrategy strategy, int off, int len)
>> dfsClient.checkOpen()
>> if (pos < getLength()) {
>>+ retries = 2
>>+ while (retries > 0) { //可以试两次
>>++ if (pos > blockEnd || currentNode == null) currentNode = blockSeekTo(pos)
```



```

//类似于 lseek
//第一次试读时 currentNode 为 null, seek 以后才确定下来
>>++ realLen = (int) Math.min(len, (blockEnd - pos + 1L))
>>++ if (locatedBlocks.isLastBlockComplete()) {
>>+++ realLen = (int) Math.min(realLen, locatedBlocks.getFileLength())
>>++ }
>>++ result = readBuffer(strategy, off, realLen, corruptedBlockMap)
//从文件读入, 如有 CRC 校验失败的坏块就记入 corruptedBlockMap
>>++> while (true) {
>>++>+ return reader.doRead(blockReader, off, len, readStatistics)
// == ByteArrayStrategy.doRead(blockReader, off, len, readStatistics)
//如果 doRead() 失败就再循环, 这里省略了异常处理的代码
>>++> } //end while
>>++ if (result >= 0) pos += result
>>++ else IOException("Unexpected EOS from the reader")
//got a EOS from reader though we expect more data on it.
>>++ if (dfsClient.stats != null) {
>>+++ dfsClient.stats.incrementBytesRead(result)
>>++ }
>>++ return result
>>+ } //end while(retries > 0)
>> } //end if if (pos < getFileLength())
>> return -1;

```

就像一般的读文件操作一样, 先是 lseek 一类的定位, 然后是从定位点开始读文件内容。所不同的是, 在 HDFS 中每次的读入都是针对具体数据块的, 因为不同的数据块可能在不同的 DataNode 节点上。所以首先要通过 blockSeekTo() 确定目标在哪一个块中。

```

[IOUtils.copyBytes() > DFSInputStream.read() > DFSInputStream.readWithStrategy()
> blockSeekTo()]

```

```

DFSInputStream.blockSeekTo(long target)
> if (target >= getFileLength()) IOException("Attempted to read past end of file")
> if (blockReader != null) { // Will be getting a new BlockReader.
>+ blockReader.close()
>+ blockReader = null
> }
> // Connect to best DataNode for desired Block, with potential offset
> while (true) {
>+ LocatedBlock targetBlock = getBlockAt(target, true) //确定读出起点在哪一个块中
>+ offsetIntoBlock = target - targetBlock.getStartOffset() //以及在目标块中的位置

```

```

>+ DNAddrPair retval = chooseDataNode(targetBlock, null)
           //从目标块各个复份所在的节点中选择一个 DataNode, 获取其地址
>+ chosenNode = retval.info //获取有关对方节点的信息
>+ InetAddress targetAddr = retval.addr //获取其网络地址
>+ StorageType storageType = retval.storageType //还有存储类型
>+ ExtendedBlock blk = targetBlock.getBlock() //获取其 ExtendedBlock 数据结构
>+ blockReader = new BlockReaderFactory(dfsClient.getConf())
           .setInetAddress(targetAddr) //节点的网络地址
           .setRemotePeerFactory(dfsClient) //DFSClient 类实现了 RemotePeerFactory 界面
           .setDatanodeInfo(chosenNode) //对方的节点
           .setStorageType(storageType) //复份的存储类型
           .setFileName(src).setBlock(blk) //块号, 虽然也有文件名, 但这不重要
           .setBlockToken(accessToken) //访问许可
           .setStartOffset(offsetIntoBlock) //有关对方节点的信息
           .setVerifyChecksum(verifyChecksum) //是否需要 CRC
           ...setUserGroupInformation(dfsClient.ugi) //访问权限
           .setConfiguration(dfsClient.getConfiguration())
           .build() == BlockReaderFactory.build()
           //构建一个 BlockReader, 并发送读块请求
>+> if (conf.shortCircuitLocalReads && allowShortCircuitLocalReads) {
           //说不定本地就存有目标块的复份, 是否允许“短路”读出, 默认为 false
>+>+ if (clientContext.getUseLegacyBlockReaderLocal()) {
>+>+> reader = getLegacyBlockReaderLocal()
>+>+> if (reader != null) return reader
>+>+ } else {
>+>+> reader = getBlockReaderLocal()
>+>+> if (reader != null) return reader
>+>+ }
>+> }
>+> if (conf.domainSocketDataTraffic) {
>+>+ reader = getRemoteBlockReaderFromDomain()
           //Get a RemoteBlockReader that communicates over a UNIX domain socket
>+>+ if (reader != null) return reader
>+> }
>+> return getRemoteBlockReaderFromTcp()
           //Get a RemoteBlockReader that communicates over a TCP socket
>+>> while (true) {
>+>>+ curPeer = nextTcpPeer()
>+>>+ if (curPeer == null) break
>+>>+ if (curPeer.fromCache) remainingCacheTries --

```

```

>+>>+ peer = curPeer.peer
>+>>+ blockReader = getRemoteBlockReader(peer)
>+>>+> if (conf.useLegacyBlockReader) {
>+>>+>+ return RemoteBlockReader.newBlockReader(fileName, block, token,
startOffset, length, ... , clientName, peer, datanode, ...)
>+>>+> } else {
>+>>+>+ return RemoteBlockReader2.newBlockReader(fileName, block, token,
startOffset, length, ... , clientName, peer, datanode, ...)
>+>>+>+> DataOutputStream out =
new DataOutputStream(new BufferedOutputStream(peer.getOutputStream()))
>+>>+>+> new Sender(out).readBlock(block, blockToken, clientName, startOffset, len,
verifyChecksum, cachingStrategy)
//首先要创建一个 Sender 对象,并让其向对方发送读块请求
>+>>+>+>> OpReadBlockProto proto = OpReadBlockProto.newBuilder()
.setHeader(DataTransferProtoUtil.buildClientHeader(blk, clientName, blockToken))
.setOffset(blockOffset).setLen(length).setSendChecksums(sendChecksum)
.setCachingStrategy(getCachingStrategy(cachingStrategy)).build() //构筑请求报文
>+>>+>+>> send(out, Op.READ_BLOCK, proto)
//向对方发送 READ_BLOCK 请求,结合上一章中 DataNode 对此请求的反应
>+>>+>+> DataInputStream in = new DataInputStream(peer.getInputStream())
>+>>+>+> status = BlockOpResponseProto.parseFrom(PBHelper.vintPrefixed(in))
>+>>+>+> checkSuccess(status, peer, block, file)
>+>>+>+> ReadOpChecksumInfoProto checksumInfo = status.getReadOpChecksumInfo()
>+>>+>+> DataChecksum checksum = DataTransferProtoUtil.fromProto(
checksumInfo.getChecksum())
>+>>+>+> firstChunkOffset = checksumInfo.getChunkOffset()
>+>>+>+> return new RemoteBlockReader2(file, block.getBlockPoolId(), block.getBlockId(),
checksum, verifyChecksum, startOffset, firstChunkOffset, len, peer,
datanodeID, peerCache)
>+>>+> }
>+>>+ return blockReader
>+>> } //end while——内层
>+>> return null
> } // end while——外层

```

DFSInputStream 已经从 NameNode 获取了目标文件的 LocatedBlocks 数组,里面有各个块在目标文件中的起点和存储地点,要知道待读出内容的起点落在哪一个块中当然不难。进一步要从存有这个目标块复份的 DataNode 中选择一个节点也是容易的事。然后就构建一个 BlockReader,让其与目标节点对接。根据不同的具体情况,这样的 BlockReader 可以有好几种,但是最典型、版本也最新的是 RemoteBlockReader2,所以我们就以此为例。

创建 BlockReader,确切地说是 RemoteBlockReader2 的时候,首先要创建一个 Sender 对

象,并让其向对方发送 READ_BLOCK 请求。而 RemoteBlockReader2,则正是为了接收对方发回的数据块复份而创建的。

回到上一层 readWithStrategy(),下一个操作就是 readBuffer(),而 readBuffer()则循环调用 ByteArrayStrategy.doRead()。

```
[IOUtils.copyBytes() > DFSInputStream.read()
> DFSInputStream.readWithStrategy() > readBuffer() > ByteArrayStrategy.doRead()]
```

```
ByteArrayStrategy.doRead(BlockReader blockReader,
                           int off, int len, ReadStatistics readStatistics)
> nRead = blockReader.read(buf, off, len) == RemoteBlockReader2.read(buf, off, len)
>> if (curDataSlice == null || curDataSlice.remaining() == 0 && bytesNeededToFinish > 0) {
>>+ readNextPacket()
>> }
>> if (curDataSlice.remaining() == 0) return -1 // we're at EOF now
>> int nRead = Math.min(curDataSlice.remaining(), len)
>> curDataSlice.get(buf, off, nRead) == ByteBuffer.get(buf, off, nRead)
>> return nRead
> updateReadStatistics(readStatistics, nRead, blockReader)
> return nRead
```

我们在前一章中看到,一个块是分成许多个 Packet 发送的,所以块的接收,即 doRead(),自然就是一个 readNextPacket()的循环:

```
[DFSInputStream.read() > DFSInputStream.readWithStrategy() > readBuffer() >
ByteArrayStrategy.doRead() > RemoteBlockReader2.readNextPacket()]

RemoteBlockReader2.readNextPacket()
> packetReceiver.receiveNextPacket(in) == PacketReceiver.receiveNextPacket(in)
>> doRead(null, in)
> PacketHeader curHeader = packetReceiver.getHeader()
> curDataSlice = packetReceiver.getDataSlice()
> if (!curHeader.sanityCheck(lastSeqNo))
    IOException("BlockReader: error in packet header " + curHeader)
> if (curHeader.getDataLen() > 0) {
>+ chunks = 1 + (curHeader.getDataLen() - 1) / bytesPerChecksum
>+ checksumsLen = chunks * checksumSize
>+ lastSeqNo = curHeader.getSeqNo()
>+ if (verifyChecksum && curDataSlice.remaining() > 0) {
>++ checksum.verifyChunkedSums(curDataSlice,
    packetReceiver.getChecksumSlice(), filename, curHeader.getOffsetInBlock())
>+ }
```

```

>+ bytesNeededToFinish -= curHeader.getDataLen()
> }
> // First packet will include some data prior to the first byte the user requested. Skip it.
> if (curHeader.getOffsetInBlock() < startOffset) {
>+ newPos = (int) (startOffset - curHeader.getOffsetInBlock())
>+ curDataSlice.position(newPos)
> }
> //If we've now satisfied the whole client read,
    //read one last packet header, which should be empty
> if (bytesNeededToFinish <= 0) {
>+ readTrailingEmptyPacket()
>+ if (verifyChecksum) {
>++ sendReadResult(Status.CHECKSUM_OK)
>++> writeReadResult(peer.getOutputStream(), statusCode) //向对方发送确认
>++>> ClientReadStatusProto.newBuilder().setStatus(statusCode).build().writeDelimitedTo(out)
                                //构筑一个报文并将其写入输出流缓冲区
>++>> out.flush() //将其发送出去
>+ } else {
>++ sendReadResult(Status.SUCCESS)
>+ }
> }

```

读过上一章中与 `readBlock()` 有关过程的读者应该很容易理解这段代码。发送方在发送 Packet 的时候可以随同数据一起发送 CRC 校验码, Packet 中的数据按一定长度分成 Chunk, 发送时(实际上从磁盘读出时就有了)按 Chunk 加上校验码, 不过校验码的加入并不影响数据的连续性, 所有的校验码都集中在一起, 而接收方则逐段加以验算。与上一章中不同的是, 将接收到的数据块作为块文件存储在本地时, 会将校验码与数据平行存储在元数据文件中, 而如果是 App 在读文件则这些校验码用过以后就不要了, 因为 App 并不关心 CRC 校验, 甚至也不知道 CRC 校验的存在。当然, 可想而知, 如果是 App 在读文件, 就没有什么 pipeline 和接力转发的问题。

总结上述读 HDFS 文件的过程, 使用者首先通过 `DistributedFileSystem.open()` 打开一个 HDFS 文件, 打开文件的结果是获得了一个输入流 `DFSInputStream`, 然后就可以对此输入流进行 `read()` 操作。

15.5 HDFS 的创建文件流程

HDFS 文件当然也有个创建的过程, `DistributedFileSystem` 提供的与创建文件有关的方法有 4 种之多:

```

class DistributedFileSystem extends FileSystem {}
] create(..., InetAddress[] favoredNodes)

```

```

] create(..., ChecksumOpt checksumOpt)
] primitiveCreate(...)
] createNonRecursive(...)

```

其中 `createNonRecursive()` 和 `primitiveCreate()` 都是简化版, 前者的特点是不递归, 只是在指定目录下创建, 如果这个目录不存在就失败, 而不会主动去创建那个目录; 后者则是“primitive”, 最简单朴素的创建。剩下的两个 `create()` 都是完全的版本了, 二者的区别在于调用参数, 其中之一增加了一个参数 `avoredNodes`, 就是希望这个文件的内容最好能存放在哪些节点上。其实这 4 个方法的基本的操作都是一样的, 读懂了其中最复杂的那个, 别的也就容易理解了。所以我们在这里选用其中最复杂最完整的一个。

```

DistributedFileSystem.create(Path f, FsPermission permission, boolean overwrite, int bufferSize,
    short replication, long blockSize, Progressable progress, InetAddress[] favoredNodes)
> Path absF = fixRelativePart(f) //首先获取目标文件的绝对路径名(全路径名)
> //解析可能存在的符号连接
> Resolver = new FileSystemLinkResolver<HdfsDataOutputStream>()
] doCall(final Path p)
    > DFSOutputStream out = dfs.create(getPathName(f), permission, ...)
        == DFSClient.create(getPathName(f), permission, ...)
    > return dfs.createWrappedOutputStream(out, statistics)
                                //返回一个 HdfsDataOutputStream 对象
] next(final FileSystem fs, final Path p)
    > if (fs instanceof DistributedFileSystem) {
    >+ myDfs = (DistributedFileSystem)fs
    >+ return myDfs.create(p, permission, overwrite, bufferSize, replication,
                                blockSize, progress, favoredNodes)
    > }
    > UnsupportedOperationException("Cannot create with" +
        " favoredNodes through a symlink to a non-DistributedFileSystem: " + f + " -> " + p)
> Resolver.resolve(this, absF) == FileSystemLinkResolver.resolve(FileSystem fileSys, Path path)

```

这里动态定义了抽象类 `FileSystemLinkResolver` 的一种扩充, 为其补上 `doCall()` 和 `next()` 两个方法, 然后创建对象并调用其 `resolve()` 函数。而 `resolve()` 则会顺着由文件路径中符号连接构成的链一直追到尽头, 直到不再是符号连接而是实际存在的文件时为止。我们在前面考察 `FsShell` 命令 `Cat` 时也遇到过符号连接的问题。这里假定不存在符号连接, 因而 `next()` 不会受到调用, 而只是考察这里的 `doCall()`。同样, 这里的 `dfs` 是个 `DFSClient` 对象, 对于 `App` 而言这就是访问 HDFS 的中介和桥梁, 目标文件的创建实际上是通过 `DFSClient.create()` 完成的, 完成之后返回一个 `DFSOutputStream` 输出流对象 `out`。不过还要进一步把这个 `out` 封装成一个 `HdfsDataOutputStream` 对象才返回给使用者。 `HdfsDataOutputStream` 是对 `FSDDataOutputStream` 的扩充, 所以这同时也是一个 `FSDDataOutputStream` 输出流。这个输出流是供写文件操作 `write()` 使用的, 而此刻我们关心的还只是文件的创建, 所以我们还是集中看 `DFSClient.create()`。

在考察这个函数的代码摘要之前, 需要先对调用参数做点说明: 第一个参数 `src` 当然是文

件路径名;第二个参数 permission 是个 FsPermission 对象,表示所创建文件对于不同用户的访问授权,这跟 Linux 文件的访问授权没有什么不同;第三个参数 flag 是一组标志位,这些标志位的定义是这样:

```
enum CreateFlag {
    CREATE((short) 0x01),          //Create a file.
                                   //如果目标文件不存在,有此标志位就加以创建,否则失败返回
    OVERWRITE((short) 0x02),       //Truncate/overwrite a file. Same as POSIX O_TRUNC
                                   //如果目标文件已经存在,有此标志位就覆盖,否则不覆盖
    APPEND((short) 0x04),          //Append to a file.
                                   //有此标志位表示允许在文件尾部添加内容(继续往下写)
    SYNC_BLOCK((short) 0x08),      //Force closed blocks to disk.
                                   //有此标志位表示强制将缓冲着的关闭数据块写入磁盘
    LAZY_PERSIST((short) 0x10);    //Create the block on transient storage (RAM) if available
                                   //If transient storage is unavailable then the block will be
                                   //created on disk.
}
```

这些标志位是互相独立的,可以合在一起使用,例如 CREATE|APPEND。

再看后面的那些参数,createParent 表示如果文件路径中的目录节点不存在就加以创建,相当于 Shell 命令行中的 -p 选项;replication 表示要求存储复份的数量;progress 用于进度报告;checksumOpt 用于 CRC 校验;favoredNodes 是个数组,列举偏好使用哪些 DataNode。

下面是 DFSClient.create() 的代码摘要:

```
[DistributedFileSystem.create() > FileSystemLinkResolver.resolve() > doCall()
> DFSClient.create()]
```

```
DFSClient . create(String src, FsPermission permission, EnumSet<CreateFlag> flag,
    boolean createParent, short replication, long blockSize, Progressable progress,
    int buffersize, ChecksumOpt checksumOpt, InetAddress[] favoredNodes)
> checkOpen()    //确认 HDFS 在运行
> if (permission == null) permission = FsPermission.getFileDefault() //默认访问权限
> FsPermission masked = permission.applyUMask(DFSClientConf.UMask)
> if (favoredNodes != null) { //如果有偏好的存储节点
>+ favoredNodeStrs = new String[favoredNodes.length]
>+ for (int i = 0; i < favoredNodes.length; i++) {
>++ favoredNodeStrs[i] = favoredNodes[i].getHostName() + ":" + favoredNodes[i].getPort()
>+ } //end for
> } //end if
> DFSOutputStream result = DFSOutputStream.newStreamForCreate(this, src, masked,
    flag, createParent, replication, blockSize, progress, buffersize,
    DFSClientConf.createChecksum(checksumOpt), favoredNodeStrs)
```

//创建 DFSOutputStream 对象

```
>>> shouldRetry = true
>>> retryCount = CREATE_RETRY_COUNT
>>> while (shouldRetry) {
>>>+ shouldRetry = false
>>>+ stat = dfsClient.namenode.create(src, masked, dfsClient.clientName,
    new EnumSetWritable<CreateFlag>(flag), createParent, replication,
    blockSize, SUPPORTED_CRYPTO_VERSIONS)
    //向 NameNode 发送请求报文,对 NameNode 上的 create()进行 RPC 调用
>>>+ break
>>> }
>>> Preconditions.checkNotNull(stat, "HdfsFileStatus should not be null!")
>>> DFSOutputStream out = new DFSOutputStream(dfsClient, src, stat,
    flag, progress, checksum, favoredNodes)
    //创建 DFSOutputStream 对象,及其内部的数据流线程
>>>> this(dfsClient, src, progress, stat, checksum)
>>>> this.shouldSyncBlock = flag.contains(CreateFlag.SYNC_BLOCK)
>>>> computePacketChunkSize(dfsClient.getConf().writePacketSize, bytesPerChecksum)
>>>> streamer = new DataStreamer(stat, traceSpan) //创建其内部的数据流线程
>>>> if (favoredNodes != null && favoredNodes.length != 0) {
>>>>+ streamer.setFavoredNodes(favoredNodes)
>>>> }
>>> out.start() //启动输出流线程 DataStreamer 的运行
>>>> streamer.start()
>>> return out
> beginFileLease(result.getFileId(), result)
>>> LeaseRenewer renewer = getLeaseRenewer()
>>>> return LeaseRenewer.getInstance(authority, ugi, this)
>>>>> LeaseRenewer r = Factory.INSTANCE.get(authority, ugi)
>>>>> r.addClient(dfsc)
>>>>> return r
>>> renewer.put(inodeId, out, this)
    == LeaseRenewer.put(long inodeId, DFSOutputStream out, DFSCClient dfsc)
>>>> if (dfsc.isClientRunning()) {
>>>>+ if (!isRunning() || isRenewerExpired()) {
>>>>++ int id = ++currentId //start a new daemon with a new id.
>>>>++ daemon = new Daemon(new Runnable() {}
    ] run()
    > LeaseRenewer.this.run(id) == LeaseRenewer.run(int id)
    > Factory.INSTANCE.remove(LeaseRenewer.this)
```

```

>>> ++ daemon.start()
>>> + }
>>> + dfsc.putFileBeingWritten(inodeId, out)
>>> +> filesBeingWritten.put(inodeId, out)
>>> +> if (lastLeaseRenewal == 0) updateLastLeaseRenewal()
>>> +>> lastLeaseRenewal = Time.now()
>>> + emptyTime = Long.MAX_VALUE
>>> }
> return result

```

笼统地说, HDFS 的创建文件操作就是三件事: 一是通过 RPC 在 NameNode 的文件目录中创建一个文件节点, 创建文件的同时也就获得了在一段时间内对此文件进行写操作的权力“租约(Lease)”; 二是在本地创建一个输出流 DFSOutputStream, 其核心是一个 DataStreamer 线程; 三是在本地创建一个 LeaseRenewer 对象, 用一个线程来管理租约, 让它不致过期。

在 Hadoop 的系统结构中, 典型的情景是文件的创建者就是文件的写入者, 谁创建谁写入, 而少有多多个写入者合写同一文件的情景, 文件共享一般只是体现在写与读之间。这是因为: 一来 Hadoop 的应用主要是数据的分析和挖掘, 一般是一次写入反复梳理, 如有输出就形成另一文件; 二来 MapReduce 的格局本来就已经由 Reducer 把很多 Mapper 产生的输出汇聚在一起, 然后才写入文件。不过, 只要文件主将目标文件的权限设置为允许, 别的用户也就可以写, 但是事先也得要通过 create(), 而不是 open() 打开文件。我们在 Cat 这个例子中看到过 open() 的使用, 那只是用于读文件, 所以不涉及租约。另外, 无论是文件主还是别的用户, 对外界的写操作只限于覆盖和添写两种; 前者是把文件中原有的内容都清除掉, 然后从头往下写; 后者是在文件末尾添加内容, 继续往下写。HDFS 不支持对文件内容的随机写访问。每当写满了一个块, 这个块就被“关闭”, 或者说已经是 Finalized, 已经封存, 从此就不允许修改了。

一旦某个用户 create() 了某个文件, 就有了写这个文件的 Lease, 到用户关闭这个文件的时候这个租约就解除了。但是用户有可能既不关闭文件也不续约, 那样租约就会因过期而作废。NameNode 上有个专管租约的 LeaseManager, 里面有个 Monitor 线程, 就是定期检查租约是否到期的, 如果过期了就会被废除。

回到 DFSCClient.create() 的 RPC 调用。DFSCClient 的这个 RPC 请求经过 PB 层和 RPC 层来到 NameNode 上, 转化成对 NameNodeRpcServer.create() 的调用:

```

[DistributedFileSystem.create() > FileSystemLinkResolver.resolve() > doCall()
> DFSCClient.create() => NameNodeRpcServer.create()]

NameNodeRpcServer.create(String src, FsPermission masked, String clientName,
    EnumSetWritable<CreateFlag> flag, boolean createParent, short replication,
    long blockSize, CryptoProtocolVersion[] supportedVersions)
> String clientMachine = getClientMachine()
> perm = new PermissionStatus(getRemoteUser().getShortUserName(), null, masked)
> fileStatus = namesystem.startFile(src, perm, clientName, clientMachine,
    flag.get(), createParent, replication, blockSize, supportedVersions)

```

```

    == FSNamesystem.startFile(src, perm, clientName, clientMachine, ...)
> return fileStatus

```

NameNodeRpcServer 只是 NameNode 上提供 RPC 服务的枢纽,具体的服务是由各种具体的模块提供的,创建文件的服务由 FSNamesystem 通过其 startFile()方法提供:

```
[DFSClient.create() => NameNodeRpcServer.create() > FSNamesystem.startFile()]
```

```

FSNamesystem.startFile(String src, PermissionStatus permissions, String holder,
    String clientMachine, EnumSet<CreateFlag> flag, boolean createParent,
    short replication, long blockSize, CryptoProtocolVersion[] supportedVersions)
> CacheEntryWithPayload cacheEntry = RetryCache.waitForCompletion(retryCache, null)
> if (cacheEntry != null && cacheEntry.isSuccess())
    return (HdfsFileStatus) cacheEntry.getPayload()
> status = startFileInt(src, permissions, holder, clientMachine, flag,
    createParent, replication, blockSize, supportedVersions, cacheEntry != null)
>> blockManager.verifyReplication(src, replication, clientMachine)
>> FSPermissionChecker pc = getPermissionChecker()
>> checkOperation(OperationCategory.WRITE)
>> byte[][] pathComponents = FSDirectory.getPathComponentsForReservedPath(src)
>> boolean create = flag.contains(CreateFlag.CREATE)
>> boolean overwrite = flag.contains(CreateFlag.OVERWRITE)
>> boolean isLazyPersist = flag.contains(CreateFlag.LAZY_PERSIST)
>> waitForLoadingFSImage()
    //防止 RPC 来得过早,如果文件系统映像尚未完成装载就在这儿睡眠等一下
>> src = resolvePath(src, pathComponents)
>> INodesInPath iip = dir.getInodesInPath4Write(src)
>> // Nothing to do if the path is not within an EZ
>> if (dir.isInAnEZ(iip)) { //目录在加密区中,略
>> }
>> ...
>> checkOperation(OperationCategory.WRITE)
>> src = resolvePath(src, pathComponents)
>> toRemoveBlocks = startFileInternal(pc, src, permissions, holder, clientMachine,
    create, overwrite, createParent, replication, blockSize,
    isLazyPersist, suite, protocolVersion, edek, logRetryCache)
    //实际的操作最终是由 startFileInternal()完成的
>> stat = dir.getFileInfo(src, false, FSDirectory.isReservedRawName(srcArg), true)
>> if (!skipSync) {
>>+ if (toRemoveBlocks != null) {
>>++ removeBlocks(toRemoveBlocks)

```

```

>> ++ toRemoveBlocks.clear()
>> + }
>> }
>> return stat
> RetryCache.setState(cacheEntry, status != null, status)
> return status

```

这个函数的核心是对 `startFileInt()` 的调用。与 `startFileInt()` 同一层上的操作,就是此前的 `RetryCache.waitForCompletion()` 和此后的 `RetryCache.setState()`,是针对有关 HDFS 文件操作的 RPC 调用进行性能上的优化。考虑到通过 RPC 进行文件操作很频繁,假如有两个完全一样的 RPC 请求接踵而至,有些操作做了一遍以后再做一遍是同样的结果(称为 idempotent,“幂等”),那就可以把前一次的结果缓存起来,后面有同样要求时就不用再做,只要返回前一次的结果就行。但是后面的请求到来时前一次操作未必就已经完成,所以这个 `RetryCache` 的安排是:开始操作前先检查一下,看是否有同样的操作已经在进行,如果是的话就睡眠等待其完成,然后直接利用其结果;如果没有,就登记一下要做的事,然后继续往前走,到操作完成之后把结果留在 `RetryCache` 中,并唤醒正在睡眠等待的线程(如果有的话)。`RetryCache` 的这套机制,是在创建 `FSNamesystem` 对象时在其构造方法中通过 `initRetryCache()` 建立起来的,但是这不属于我们关心的问题,这里就不深入进去了。

我们关心的是,在 `startFileInt()` 内部,最终的操作是由 `startFileInternal()` 完成的:

```

[DFSClient.create() => NameNodeRpcServer.create() > FSNamesystem.startFile()
> startFileInt() > startFileInternal()]

```

```

startFileInternal(FSPermissionChecker pc, String src, PermissionStatus permissions,
    String holder, String clientMachine, boolean create, boolean overwrite,
    boolean createParent, short replication, long blockSize, boolean isLazyPersist,
    CipherSuite suite, CryptoProtocolVersion version, EncryptedKeyVersion edek,
    boolean logRetryEntry)
> INodesInPath iip = dir.getINodesInPath4Write(src) //dir 是个 FSDirectory 对象
> INode inode = iip.getLastINode() //获取路径中的最后一个 INode
> if (dir.isInAnEZ(iip)) { //目录在加密区中,略
> }
> INodeFile myFile = INodeFile.valueOf(inode, src, true) //这是要创建的目标文件
> if (isPermissionEnabled) {
> + if (overwrite && myFile != null) checkPathAccess(pc, src, FsAction.WRITE)
//如果目标文件存在,且要求 overwrite,则检查写文件权限
> + checkAncestorAccess(pc, src, FsAction.WRITE)
> }
> if (!createParent) verifyParentDir(src)
> try {
> + if (myFile == null) { //如果目标文件不存在,那就必须有 CREATE 标志才能创建

```

```

> ++ if (!create) FileNotFoundException(           //没有 CREATE 标志就不允许创建
    "Can't overwrite non-existent " + src + " for client " + clientMachine)
    // 如果 create == true, 则说明有 CREATE 标志, 可以创建目标文件
> + } else { //目标文件原已存在
> ++ if (overwrite) { //如果要求 OVERWRITE, 就把原有同名文件删掉, 然后重新创建
> +++ toRemoveBlocks = new BlocksMapUpdateInfo()
> +++ List<INode> toRemoveINodes = new ChunkedArrayList<INode>()
> +++ ret = dir.delete(src, toRemoveBlocks, toRemoveINodes, now())
> +++ if (ret >= 0) {
> ++++ incrDeletedFileCount(ret)
> ++++ removePathAndBlocks(src, null, toRemoveINodes, true)
> +++ }
> ++ } else { //否则, 如果没有加 OVERWRITE, 则发起异常
> +++ // If lease soft limit time is expired, recover the lease
> +++ recoverLeaseInternal(myFile, src, holder, clientMachine, false)
> +++ FileAlreadyExistsException(src + " for client " + clientMachine + " already exists")
    //因为异常, 就跳过了目标文件的创建
> ++ }
> + }
> + checkFsObjectLimit()
> + Path parent = new Path(src).getParent()
> + if (parent != null && mkdirsRecursively(parent.toString(), permissions, true, now())) {
    //也许需要先往下创建一系列子目录, 然后才是目标文件
> ++ INodeFile newNode = dir.addFile(src, permissions, replication,
    blockSize, holder, clientMachine)
    //为目标文件创建一个 INodeFile 对象, 并将其添加到 FSDirectory 的目录树中
    //这就完成了目标文件的创建, 当然此时还是个空文件
> + }
> + if (newNode == null) IOException("Unable to add " + src + " to namespace")
    //如果失败就发起异常
> + leaseManager.addLease(newNode.getFileUnderConstructionFeature().getClientName(), src)
    == LeaseManager.addLease(String holder, String src) //把目标文件加到该用户的 Lease 中
> + Lease lease = getLease(holder) //找到该用户的租约
> + if (lease == null) {
> ++ lease = new Lease(holder) //如果还没有就创建一个新的租约
> ++ leases.put(holder, lease) //并把它放在租约的便查 Map 即 leases 中
> ++ sortedLeases.add(lease) //同时也放进按时间排序的租约集合 sortedLeases 中
> + } else {
> ++ renewLease(lease) //如果该用户原来就有租约, 就办一下续约
> + }

```



```

>+> sortedLeasesByPath.put(src, lease) //把目标文件加入租约中
>+> lease.paths.add(src) //也加入这个可以按路径查询的树状集合中
>+> return lease
>+ setNewINodeStoragePolicy(newNode, iip, isLazyPersist)
>+ // record file record in log, record new generation stamp
>+ getEditLog().logOpenFile(src, newNode, overwrite, logRetryEntry)
    == FSEditLog.logOpenFile(String path, INodeFile newNode,
                                boolean overwrite, boolean toLogRpcIds)
    //记入文件系统操作(更改)日志

>+> ...
>+> logEdit(op)
>+ return toRemoveBlocks
> } catch (IOException ie) {
>+ NameNode.stateChangeLog.warn(
    "DIR * NameSystem.startFile: " + src + " " + ie.getMessage())
>+ throw ie
> }

```

这里对目标文件节点的处理、对租约的处理乃至对日志的处理都是放在一个 try{}catch() 结构中的, 程序中像这样的措施是常态, 为不致冲淡主题, 我们在摘要中一般都加以省略, 但是这里的处理却值得注意。我们着重看一种情景, 就是要创建的目标文件 myFile 原已存在的情况下该如何处理: 如果调用参数中指定了 OVERWRITE, 就意味着把原来的文件删掉, 重新创建一个。但是, 如果用户没有指定要 OVERWRITE, 那就会发起一个 FileAlreadyExistsException 异常, 这是对 IOException 的扩充, 因而会被下面的 catch 所捕捉, 这样就跳过了创建目标文件的那段代码, 创建文件就失败了。所以, 对于一个已经存在的文件, 别人是无法通过 create() 加以打开, 然后在文件末尾再继续添加内容的; OVERWRITE 意味着全部从头来起, 实际上只是保留了文件的路径不变。

但是 App 可以通过 HDFS 的中介 DFSCClient 发起对文件的 append() 操作, 这个操作相当于打开文件用于在尾部添加内容, 这也是一种打开文件操作:

```

DFSCClient.append(String src, int buffersize, Progressable progress)
> checkOpen()
> DFSOutputStream result = callAppend(src, buffersize, progress)
>> lastBlock = namenode.append(src, clientName) //对 NameNode 的 RPC 调用
    //namenode 是通往 NameNode 的 proxy
>> HdfsFileStatus newStat = getFileInfo(src)
>> return DFSOutputStream.newStreamForAppend(this, src,
    buffersize, progress, lastBlock, newStat, dfsClientConf.createChecksum())
> beginFileLease(result.getFileId(), result)
> return result //result 是个 DFSOutputStream 对象

```

这个函数返回一个 DFSOutputStream 对象, 这个输出流可用于在文件尾部添加内容。显

然,这里的核心是经由 `callAppend()` 所做的对 `NameNode` 的 RPC 调用。

这里顺便说一下,既然是对 `NameNode` 的 RPC 调用,那么这个 `DFSClient` 是在 `DataNode` 上? 还是也在 `NameNode` 上? 其实,它可以在集群内的任何节点上,可以在某个 `DataNode` 所在的节点上,也可以在 `NameNode` 所在的节点上。但是它既不在 `DataNode` 中,也不在 `NameNode` 中。即使物理上是与 `NameNode` 同在一个节点上,逻辑上它也是在另外一个 JVM 中,所以仍需要 RPC。

那么 `NameNode` 在收到 `addBlock` 的请求报文时做些什么呢? 我们看代码摘要:

```
[DFSClient.append() > callAppend() => NameNodeRpcServer.append()]

NameNodeRpcServer.append(String src, String clientName)
> String clientMachine = getClientMachine()
> LocatedBlock info = namesystem.appendFile(src, clientName, clientMachine)
  == FSNamesystem.appendFile(src, clientName, clientMachine)
>> CacheEntryWithPayload cacheEntry = RetryCache.waitForCompletion(retryCache, null)
>> if (cacheEntry != null && cacheEntry.isSuccess()) {
>>+ return (LocatedBlock) cacheEntry.getPayload()
>> }
>> lb = appendFileInt(src, holder, clientMachine, cacheEntry != null)
>>> FSPermissionChecker pc = getPermissionChecker() //用于访问权限检查
>>> checkOperation(OperationCategory.WRITE)
>>> byte[][] pathComponents = FSDirectory.getPathComponentsForReservedPath(src)
>>> checkNameNodeSafeMode("Cannot append to file" + src)
>>> src = resolvePath(src, pathComponents) //通过解析路径得到最终的目标文件名
>>> lb = appendFileInternal(pc, src, holder, clientMachine, logRetryCache)
>>> logAuditEvent(true, "append", srcArg)
>> success = true
>> RetryCache.setState(cacheEntry, success, lb)
>> return lb
> return info
```

如前所述, `NameNodeRpcServer` 只是相当于一个交通枢纽和集散中心,具体的服务是由 `FSNamesystem` 通过 `appendFile()` 提供的。遵循同样的程序模式,这里也是在 `appendFileInt()` 的外面套了一层 `RetryCache` 的优化。而 `appendFileInt()`, 则通过 `appendFileInternal()` 完成实际的操作:

```
[NameNodeRpcServer.append() > FSNamesystem.appendFile() > appendFileInternal()]
```

```
FSNamesystem.appendFileInternal(FSPermissionChecker pc, String src,
                                String holder, String clientMachine, boolean logRetryCache)
中 > INodesInPath iip = dir.getInodesInPath4Write(src)
量 > INode inode = iip.getLastINode()
```

```

> if (inode == null) FileNotFoundException("failed to append to non-existent file " ...)
> INodeFile myFile = INodeFile.valueOf(inode, src, true) //获取目标文件的 INode
> BlockStoragePolicy lpPolicy = blockManager.getStoragePolicy("LAZY_PERSIST")
> if (lpPolicy != null && lpPolicy.getId() == myFile.getStoragePolicyID()) {
>+ UnsupportedOperationException("Cannot append to lazy persist file " + src)
> }
> // Opening an existing file for write - may need to recover lease.
> recoverLeaseInternal(myFile, src, holder, clientMachine, false)
> myFile = INodeFile.valueOf(dir.getInode(src), src, true)
    //再次获取目标文件的 INode(可能已因 recoverLeaseInternal()引起变化)
> BlockInfo lastBlock = myFile.getLastBlock()
    //从 INodeFile 对象中获取其最后一个数据块的 BlockInfo
> // Check that the block has at least minimum replication.
> if(lastBlock != null && lastBlock.isComplete()
    && !getBlockManager().isSufficientlyReplicated(lastBlock)) {
>+ IOException("append: lastBlock = " + lastBlock + " of src = " + src +
    " is not sufficiently replicated yet.")
> }
> return prepareFileForWrite(src, myFile, holder, clientMachine, true,
    iip.getLatestSnapshotId(), logRetryCache)
>> file.recordModification(latestSnapshot)
>> INodeFile cons = file.toUnderConstruction(leaseHolder, clientMachine)
    //将目标文件的状态改成正在施工,即 UnderConstruction
>> leaseManager.addLease(cons.getFileUnderConstructionFeature().getClientName(), src)
    //将目标文件加进该用户的租约
>> LocatedBlock ret = blockManager.convertLastBlockToUnderConstruction(cons)
    //最后一个数据块的状态也变成 UnderConstruction
>> if (ret != null) { //update the quota: use the preferred block size for UC block
>>+ long diff = file.getPreferredBlockSize() - ret.getBlockSize()
>>+ dir.updateSpaceConsumed(src, 0, diff * file.getBlockReplication())
>> }
>> if (writeToEditLog) {
>>+ getEditLog().logOpenFile(src, cons, false, logRetryCache) //记入操作日志
    == FSEditLog.logOpenFile(String path, INodeFile newNode,
        boolean overwrite, boolean toLogRpcIds)
>> }
>> return ret

```

摘要中加了注释,读者应该不难理解。

这里对最后的日志操作加点说明。我们在前面因 create()而导致的 startFileInternal()中看到,那里的日志操作是 logOpenFile(src, newNode, overwrite, logRetryEntry);而这里是

由 `append()` 导致的 `appendFileInternal()` $>$ `prepareFileForWrite()`, 但是其日志操作同样也是 `logOpenFile(src, cons, false, logRetryCache)`。这充分说明 `create()` 和 `append()` 这两种操作在某种意义上是等价的。所不同的是: 前者的第二个参数是 `newNode`, 那就是新创建的 `INodeFile`; 而后者是 `cons`, 这是原来已经存在并已关闭而现在改成了 `UnderConstruction`, 又可以往里写的 `INodeFile`。还有, 前者的第三个参数是变量 `overwrite`, 可以是 `true` 也可以是 `false`; 而后者则固定为 `false`。这也很好理解, 那是别人的文件, 虽然允许你添加内容, 你又怎么可以覆盖重写呢?

所以, App 之调用 `DFSClient.append()`, 说到底就是 `prepareFileForWrite()`, 这个函数名很好地揭示了 `append()` 的实质。

最后, 回到 App 这一边 `DFSClient.append()` 的代码摘要中, 通过 `newStreamForAppend()` 创建的显然也是个 `DFSOutputStream`, 这也跟在 `DFSClient.create()` 中创建的输出流一样, 后面对 `beginFileLease()` 的调用也是一样。

15.6 文件租约

如上所述, 当一个 Client 在 HDFS 文件系统中创建一个文件时, 它就获得了在一定时间内对此文件的“租约 (Lease)”, 从而就有了对此文件进行写操作的权利, 同时也可防止别的 Client 也来写同一文件, 所以租约的实质是对于文件写入权的独占, 拿到了对于一个文件的租约就相当于对此文件加上了 (针对其他用户的) 防写锁。但是租约是有时间限制的, 一旦过期, 创建者就失去了写文件的权利。下面是 `Lease` 类定义的摘要:

```
class LeaseManager.Lease implements Comparable<Lease> {}
] String holder           //租约的持有者
] long lastUpdate         //最近一次更新的时间
] Collection<String> paths //该持约者所租有的所有文件
] Lease(String holder)
    > this.holder = holder
    > renew()
] renew()
    > this.lastUpdate = now()
] removePath(String src)
    > return paths.remove(src)
```

可见, 所谓租约即 `Lease` 对象并不是每个文件一份, 而是一个用户一份, 类似于一个账户, 在同一份租约中可以包含多个文件。

租约有时间限制, 一旦过期创建者就失去了写文件的权利, 所以文件的租用者需要注意不让租约过期, `LeaseRenewer` 中的线程就是专门管这个事。

```
class LeaseRenewer{}
] long renewal = HdfsConstants.LEASE_SOFTLIMIT_PERIOD/2
] Daemon daemon //A daemon for renewing lease
```

```

] List<DFSClient> dfsclients // DFSClient
] class Factory {} //A factory for sharing LeaseRenewer objects among DFSClient instances
]] Map<Key, LeaseRenewer> renewers //A map for per user per namenode renewers.
] addClient(final DFSClient dfsc)
] put(final long inodeId, final DFSOutputStream out, final DFSClient dfsc)
  > if (dfsc.isClientRunning()) {
  >+ if (!isRunning() || isRenewerExpired()) {
  >++ id = ++currentId
  >++ daemon = new Daemon(new Runnable())
  ] run()
  > LeaseRenewer.this.run(id)
  >++ daemon.start()
  >+ }
  > }

```

DFSClient 在 create() 和 append() 的末尾都要通过 DFSClient.beginFileLease() 调用 LeaseRenewer.put(), 将目标文件的 FileId 和所创建的 DFSOutputStream 添加到 DFSClient 内部的一个 MAP, 即集合 filesBeingWritten 中去。如果尚未创建 LeaseRenewer 对象就加以创建, 包括其守护线程。当然, 关闭文件时则要通过 endFileLease() 加以删除。LeaseRenewer 对象既然是由 DFSClient 所建, 就理所当然属于 DFSClient, 最终属于具体的 App 进程。另外, LeaseRenewer 是针对具体 NameNode 的, 所以在联邦制的集群中同一 App 进程内部可以有多个 LeaseRenewer。

LeaseRenewer 的守护线程负责办理租约延期:

```

LeaseRenewer.daemon.run(int id) //LeaseRenewer 是用户方的线程
> for(long lastRenewed = Time.now(); !Thread.interrupted();
    Thread.sleep(getSleepPeriod())) { //每轮循环之后都要睡一会儿
  >+ long elapsed = Time.now() - lastRenewed
  >+ if (elapsed >= getRenewalTime()) { //如果租约到期
  >++ renew() //续约
  >++> List<DFSClient> copies = new ArrayList<DFSClient>(dfsclients)
  >++> Collections.sort(copies, new Comparator<DFSClient>())
  >++> for(int i = 0; i < copies.size(); i++) {
  >++>+ DFSClient c = copies.get(i) //LeaseRenewer 直接知道的是某个 DFSClient 对象
    //但是从 DFSClient 对象可知其属于哪个用户(Client)
  >++>+ if (!c.getClientName().equals(previousName)) { //如果这是不同于先前的另一个用户
  >++>+ c.renewLease() = DFSClient.renewLease() //为这个 DFSClient 对象办理续约
  >++>+> namenode.renewLease(clientName) //通过 RPC 要求 NameNode 为此用户续约
  >++>+> updateLastLeaseRenewal()
  >++>+> return true
  >++>+> previousName = c.getClientName()

```

```

> ++> + }
> ++> }
> ++ lastRenewed = Time.now()
> + }
> + if (id != currentId || isRenewerExpired()) {
> ++ return
> + }
> + if (!clientsRunning() && emptyTime == Long.MAX_VALUE) emptyTime = Time.now()
> }

```

用户方的线程 LeaseRenewer.daemon 过一阵就通过 RPC 调用一次 NameNode 上的 renewLease(), 要求延续其租约:

```

NameNodeRpcServer.renewLease(String clientName)
> namesystem.renewLease(clientName) == FSNamesystem.renewLease(String holder)
>> checkOperation(OperationCategory.WRITE)
>> checkNameNodeSafeMode("Cannot renew lease for " + holder)
>> leaseManager.renewLease(holder) == LeaseManager.renewLease(String holder)
>>> renewLease(getLease(holder)) //这个 holder 就是上一层中的 clientName
>>>> if (lease != null) {
>>>>+ sortedLeases.remove(lease)
>>>>+ lease.renew() == Lease.renew()
>>>>+> this.lastUpdate = now() //Only LeaseManager object can renew a lease
>>>>+ sortedLeases.add(lease)
>>>> }

```

而 NameNode 上的 LeaseManager, 则有个线程 Monitor, 过一阵就检查一下是否有租约已经过期, 如果有就将其废除:

```

LeaseManager.Monitor.run()
> for(; shouldRunMonitor && fsnamesystem.isRunning(); ) {
> + if (!fsnamesystem.isInSafeMode()) needSync = checkLeases()
> +> needSync = false
> +> for(; sortedLeases.size() > 0; ) {
> +>+ Lease oldest = sortedLeases.first() //从按时间排序的集合中取最老的那份合约
> +>+ if (!oldest.expiredHardLimit()) return needSync //最老的也没过期, 就不用再往下看了
> +>+ LOG.info(oldest + " has expired hard limit")
> +>+ String[] leasePaths = new String[oldest.getPaths().size()] //那份租约中有几个文件
> +>+ oldest.getPaths().toArray(leasePaths)
> +>+ for(String p : leasePaths) {
> +>+> try {
> +>+>+ boolean completed = fsnamesystem.internalReleaseLease(oldest, p,
HdfsServerConstants.NAMENODE_LEASE_HOLDER)

```



```

>+>+++ if (!needSync &&!completed) needSync = true
>+>++ } catch (IOException e) {
>+>+++ LOG.error("Cannot release the path " + p + " in the lease " + oldest, e)
>+>+++ removing.add(p)
>+>++ } //end try{}catch()
>+>++ for(String p : removing) removeLease(oldest, p)
>+>++> sortedLeasesByPath.remove(src) //从 sortedLeasesByPath 中去除这个文件
>+>++> lease.removePath(src) //从这个 Lease 中去除这个文件
>+>++> if (!lease.hasPath()) { //要是这个 Lease 中不再有别的文件
>+>++>+ leases.remove(lease.holder) //就从租约集合中去除这个租约
>+>++>+ sortedLeases.remove(lease) //从 sortedLeases 中也去除这个租约
>+>++> }
>+>+ } //end for(String p : leasePaths)
>+> } //end for(; sortedLeases.size() > 0; )
>+> return needSync
>+ if (needSync) fsnamesystem.getEditLog().logSync()
>+ Thread.sleep(HdfsServerConstants.NAMENODE_LEASE_RECHECK_INTERVAL) //2000,2 秒
> }

```

这样,就可以防止有人拿到租约以后很长时间不用,而别人要用又拿不到的情况发生。比方说,一个节点上的 App 拿到了若干个文件的租约,但是忽然这个节点的网线断了,这时候 NameNode 上的 LeaseManager 就收不到来自这个节点的 renew 请求,过一会儿相应的租约就都到期而被解除。要不然这些文件就永远被锁住了。

15.7 HDFS 的写文件流程

通过 DFSCClient.create()创建了目标文件,或通过 DFSCClient.append()打开了目标文件,并取得了对这个文件的租约,就可以写文件了。不过 HDFS 文件系统并不提供我们一般常见于其他文件系统中的 write(),而只提供 addBlock(),就是在文件的末尾添加数据块。不过 DFSClien 也并不是将此 addBlock()直接提供给用户,而是如前所述为 App 提供了一个 DFSOutputStream 输出流,让 App 可以通过对输出流的 write()操作实现对于 HDFS 文件的写入。这样,DFSOutputStream 输出流为用户提供的操作界面就与别的文件系统一致了。

我们在前面看到,DFSCClient 受 App 调用,通过 create()创建文件或通过 append()要求对文件添加内容时,如果一切正常,就会为此文件创建一个 DFSOutputStream 对象,再将其封装在一个 HdfsDataOutputStream 对象内部,才把后者返回给使用者。但是当然,对 HdfsDataOutputStream 对象的操作最终会被转嫁落实到 DFSOutputStream 对象上。我们知道这个 HdfsDataOutputStream 是对 FSDataOutputStream 的扩充,所以这同时也是一个 FSDataOutputStream 输出流。进一步,这归根结底是个输出流,所以 DFSCClient.create()的返回类型为 OutputStream。但正是 DFSOutputStream 体现了 HDFS 的特殊性,所以我们有必要为这个类做一个摘要:

```

class DFSOutputStream extends FSOutputSummer implements ...{
    ] DFSClient dfsClient
    ] ByteArrayManager byteArrayManager
    ] Socket s
    ] String src //目标文件路径
    ] long fileId //打开文件号
    ] long blockSize //块的大小
    ] LinkedList<Packet> dataQueue //需要往外发出的数据 Packet 队列
    ] LinkedList<Packet> ackQueue //来自对方的响应 Packet 队列
    ] DataStreamer streamer //数据输出流线程
    ] start()
    >streamer.start()

```

这种输出流的一个重要成分是 dataQueue,这是一个以 Packet 为元素的 LinkedList。凡是有需要发送出去的 Packet,就将其挂在这个输出流的 dataQueue 中,我们知道数据块是分成 Packet 往外发送的。更为关键的成分是 DataStreamer 线程 streamer。如果说输出流是 stream,那么这个线程就是 stream 的形成者 streamer,输出流是由它产生和形成的,凡是挂入 dataQueue 中的 Packet 都由这个线程发送出去。我们先看一下 DataStreamer 数据结构部分的摘要:

```

class DataStreamer extends Daemon {
    ] ExtendedBlock block; // its length is number of bytes acked
    ] Token<BlockTokenIdentifier> accessToken
    ] DataOutputStream blockStream //数据块的输出流
    ] DataInputStream blockReplyStream //对方应答信息的输入流
    ] ResponseProcessor response //应答信息的处理器
    ] DatanodeInfo[] nodes //用来存储数据块复份的 DataNode 节点
    ] StorageType[] storageTypes //各个复份的存储类型
    ] String[] storageIDs //用来存储各个复份的设备
    ] LoadingCache<DatanodeInfo, DatanodeInfo> excludedNodes //应该避免使用的节点
    ] String[] favoredNodes //希望使用的节点
    ] setPipeline(LocatedBlock lb) //设置数据块存储流水线
    ] run()

```

上面我们看到的是这两个类的数据结构部分的摘要,下面再看它们所提供的操作和所起的作用。

如前所述,DFSClient.create()和 DFSClient.append()这二者最后都会创建并返回一个 DFSOutputStream 对象,这个对象中有个部件 DataStreamer 是个线程。这个过程其实并不简单,里面还包括向 NameNode 所指定的一组 DataNode 发送 WRITE_BLOCK 请求并建立数据块传输流水线的过程。DFSOutputStream 类有三个构造函数,其中的两个分别用于 create()和 append(),二者大同小异,我们就看用于 append()的这个 DFSOutputStream():

```
[DFSClient.callAppend() > DFSOutputStream.newStreamForAppend() > DFSOutputStream()]
```

```
DFSOutputStream(DFSClient dfsClient, String src, Progressable progress,
    LocatedBlock lastBlock, HdfsFileStatus stat, DataChecksum checksum)
    // Construct a new output stream for append.
> this(dfsClient, src, progress, stat, checksum)
> initialFileSize = stat.getLen(); // length of file when opened
> if (lastBlock != null) { // 最后一块未填满
>+ bytesCurBlock = lastBlock.getBlockSize()
    // indicate that we are appending to an existing block
>+ streamer = new DataStreamer(lastBlock, stat, bytesPerChecksum, traceSpan)
> } else { // 另起新块:
>+ computePacketChunkSize(dfsClient.getConf().writePacketSize, bytesPerChecksum)
>+ streamer = new DataStreamer(stat, traceSpan)
>+ isAppend = false
>+ isLazyPersistFile = isLazyPersist(stat)
>+ stage = BlockConstructionStage.PIPELINE_SETUP_CREATE // 不是 PIPELINE_SETUP_APPEND
>+ traceSpan = span
> } // end if
> this.fileEncryptionInfo = stat.getFileEncryptionInfo()
```

比之因 create() 操作而创建 DFSOutputStream 对象, 因 append() 操作而创建时有个特殊的考虑, 就是原先的最后一块可能尚未填满, 由此而来的处理自然会复杂一些, 而且在创建 DataStreamer 对象时所调用的构造函数也因此而不同。这里摘要中展开的是比较简单的那一个, 就是另起新块, 无须考虑最后一块尚未填满时所用的构造函数。值得注意的是, 尽管是因 append() 而创建, 这里把数据块传输的阶段设置成 PIPELINE_SETUP_CREATE, 而不是 PIPELINE_SETUP_APPEND, 因为此时无须考虑怎样写入尚未填满的最后一块。

这里的变量 stage, 是个 BlockConstructionStage 类对象, 这是个枚举类型, 我们在前面已经看到过它的定义。

DataStreamer 对象初创时在其构造函数中把 stage 设置成 PIPELINE_SETUP_CREATE 或 PIPELINE_SETUP_APPEND, 这要看其所在的 DFSOutputStream 输出流是因何而建, 被调用的是哪一个构造函数(参数的个数不同)。

创建了 DFSOutputStream 对象之后, 在 newStreamForAppend() 中又调用它的 start() 方法, 从前面 DFSOutputStream 的摘要中可见这就是对 streamer.start(), 即 DataStreamer.start() 的调用。DataStreamer 是对 Daemon 的扩充, 自己又没有定义 start() 方法, 所以那就是从 Daemon 继承的 start(), 那会调用线程的 run() 函数, 就是 DataStreamer.run():

```
[DFSClient.callAppend() > DFSOutputStream.newStreamForAppend()
> DFSOutputStream.start() > DataStreamer.start() > DataStreamer.run()]
```

```
DFSOutputStream.DataStreamer.run()
```

```

> while (!streamerClosed && dfsClient.clientRunning) {
>+ // if the Responder encountered an error, shutdown Responder
>+ if (hasError && response != null) {
>++ response.close(); response.join(); response = null;
>+ }
>+ if (hasError && (errorIndex >= 0 || restartingNodeIndex >= 0)) {
>++ boolean doSleep = processDatanodeError()
    //如果发送过程中有错,或有目标节点重启,使已经发送并移入 ackQueue 的 Packet
    //没有得到应答确认,就把 ackQueue 队列中的 Packet 移回 dataQueue,以供重发
    //但是这个过程可能有点长,也许需要睡眠等待
>+ }
>+ // wait for a packet to be sent.
>+ while ((!streamerClosed && !hasError && dfsClient.clientRunning &&
    dataQueue.size() == 0 && (stage != BlockConstructionStage.DATA_STREAMING
    || stage == BlockConstructionStage.DATA_STREAMING &&
    now - lastPacket < dfsClient.getConf().socketTimeout/2)) || doSleep) {
>++ dataQueue.wait(timeout)
    //睡眠等待直至队列中有 Packet,或上列条件不再满足,包括超时
    //stage 是 DataStreamer 内部的一个 BlockConstructionStage 对象
>+ } //end while,循环直至 dataQueue 队列中有了 Packet,或不再满足循环条件
>+ doSleep = false
>+ if (streamerClosed || hasError || !dfsClient.clientRunning) continue
>+ if (dataQueue.isEmpty()) {
>++ Packet one = createHeartbeatPacket() //实在没有 Packet 可发就创建一个心跳 Packet
>++> buf = new byte[PacketHeader.PKT_MAX_HEADER_LEN]
>++> return new Packet(buf, 0, 0, Packet.HEART_BEAT_SEQNO, getChecksumSize())
    //参数 chunksPerPkt 为 0,表示没有数据
>+ } else {
>++ Packet one = dataQueue.getFirst(); // regular data packet,有 Packet 待发送,就用之
>+ }
    //有了一个 Packet,就可进入 DATA_STREAMING 阶段
    -----
>+ if (stage == BlockConstructionStage.PIPELINE_SETUP_CREATE) {
    //见前,输出流是因 create()而建,或无须考虑最后一块尚未填满:
>++ DFSCliet.LOG.debug("Allocating new block")
>++ lb = nextBlockOutputStream()//这是重要的一步,后面另行展开
    //向一 DataNode 发出 WRITE_BLOCK 请求,建立 Pipeline
    //每个 Block 都有一个独立的 Pipeline,成为一个独立的 Stream
>++ setPipeline(lb) //设置这个 Block 的 Pipeline:
>++> setPipeline(lb.getLocations(), lb.getStorageTypes(), lb.getStorageIDs())

```

```

>+>> this.nodes = nodes
>+>> this.storageTypes = storageTypes
>+>> this.storageIDs = storageIDs
>+ initDataStreaming() //数据输出流的初始化:
>+> response = new ResponseProcessor(nodes) //创建响应处理线程
>+> response.start() //并启动之
>+> stage = BlockConstructionStage.DATA_STREAMING //进入 DATA_STREAMING 阶段
>+ }
>+ else if (stage == BlockConstructionStage.PIPELINE_SETUP_APPEND) {
    //输出流是因 append()而建,并须考虑最后一块尚未填满
>+> DFSClient.LOG.debug("Append to block " + block)
>+> setupPipelineForAppendOrRecovery()
    //向一 DataNode 发出 WRITE_BLOCK 请求,建立 Pipeline
>+> initDataStreaming() //见上,也进入 DATA_STREAMING 阶段
>+ } //end if
-----
>+ lastByteOffsetInBlock = one.getLastByteOffsetBlock() //这就是要发送的 Packet
>+ if (one.lastPacketInBlock) { //如果这是数据块中的最后一个 Packet
>+> while (!streamerClosed && !hasError && ackQueue.size() != 0
    && dfsClient.clientRunning) {
    //wait for all data packets have been successfully acked
    //如果 ackQueue 队列中还有 Packet,就说明发送后还没有得到回应
>+>> dataQueue.wait(1000) //wait for acks to arrive from datanodes
>+> } //end while
    //当程序到达这一点时,同一块中此前发送的 Packet 均已得到对方回应
>+> if (streamerClosed || hasError || !dfsClient.clientRunning) continue
>+> stage = BlockConstructionStage.PIPELINE_CLOSE
>+ } //end if (one.lastPacketInBlock)
>+ // send the packet
>+ if (!one.isHeartbeatPacket()) { //move packet from dataQueue to ackQueue
>+> dataQueue.removeFirst() //虽然这个 Packet 已在我们手上,但尚未从 dataQueue 中摘除
>+> ackQueue.addLast(one) //把这 Packet 挂到 ackQueue 队列中,等待回应
>+> dataQueue.notifyAll()
>+ }
>+ // write out data to remote datanode,这是实际的发送
>+ one.writeTo(blockStream) == Packet.writeTo(blockStream) //将这个 Packet 写入 blockStream
    // blockStream 是 DataStreamer 类的内部成分,是个 DataOutputStream 类对象
    //创建于前面的 nextBlockOutputStream()或 setupPipelineForAppendOrRecovery()中
    //注意,这个 Packet 也许是个数据 Packet,也许是个 HeartbeatPacket
>+ blockStream.flush() == DataOutputStream.flush()

```

```

>+ long tmpBytesSent = one.getLastByteOffsetBlock()
>+ if (bytesSent < tmpBytesSent) bytesSent = tmpBytesSent
>+ if (streamerClosed || hasError || !dfsClient.clientRunning) continue
>+ if (one.lastPacketInBlock) {    //Is this block full?
>+ while (!streamerClosed &&!hasError && ackQueue.size() != 0
&& dfsClient.clientRunning) {
>+ dataQueue.wait(1000); // wait for acks to arrive from datanodes
>+ }
>+ if (streamerClosed || hasError || !dfsClient.clientRunning) continue
>+ endBlock()
>+ } //end if
>+ if (artificialSlowdown != 0 && dfsClient.clientRunning) Thread.sleep(artificialSlowdown)
> } //end while(!streamerClosed && dfsClient.clientRunning), 回过去试图发送下一个 Packet
> closeInternal()

```

这个线程的主循环总是想要从 dataQueue 队列中得到一个待发送的 Packet, 或者因为相隔时间太长就自己生成一个心跳 Packet, 如果此时流水线 Pipeline 所处的阶段还在 PIPELINE_SETUP_CREATE 或 PIPELINE_SETUP_APPEND 这二者之一(说明这是第一个 Packet, 因为 DataStreamer 对象在其构造函数中就被设置成这二者之一), 就分别调用 nextBlockOutputStream() 或 setupPipelineForAppendOrRecovery(); 并调用 initDataStreaming(), 使 Pipeline 进入 DATA_STREAMING 阶段, 就是在 Pipeline 中传送数据块的阶段。反之, 如果不在这二者之一, 例如已经进入了 DATA_STREAMING 阶段, 那就会跳过这两个阶段的处理, 直接就从事 DATA_STREAMING 阶段的处理。

这两个函数的目的是一样的, 都是要为 Pipeline 建立起一个输出流, 所不同的是 setupPipelineForAppendOrRecovery() 较为复杂一些, 因为目标文件中的最后一个数据块很可能尚未写满, 得要在这后面添加上去。不过虽说是多个节点的 Pipeline, 这里只要与其中的第一个节点建立起连接就行, 后面它们自会逐站转发, 这我们已经在上一章中看到过了。限于篇幅, 我们在这里只是看一下 nextBlockOutputStream(), 把这个搞懂, 另一个也就不难理解了。

```
[DFSOutputStream.DataStreamer.run() > nextBlockOutputStream()]
```

```

DFSOutputStream.DataStreamer.nextBlockOutputStream()
> count = dfsClient.getConf().nBlockWriteRetry //获取重试次数
> oldBlock = block
> do {
>+ DatanodeInfo[] excluded = excludedNodes.getAllPresent(
excludedNodes.asMap().keySet()).keySet().toArray(new DatanodeInfo[0])
>+ block = oldBlock
>+ lb = locateFollowingBlock(startTime, excluded.length > 0?excluded : null)
>+ block = lb.getBlock()

```



```

>+ block.setNumBytes(0)
>+ nodes = lb.getLocations()
>+ storageTypes = lb.getStorageTypes()
>+ // Connect to first DataNode in the list.
>+ success = createBlockOutputStream(nodes, storageTypes, 0L, false)
> } while (!success && --count >= 0) //循环若干次,具体次数取决于 count
> return lb

```

这是比较简单的情景,这里就是两个操作:首先是 `locateFollowingBlock()`,目的是要知道下一个数据块应该去向哪些节点,孰先孰后;然后就是 `createBlockOutputStream()`,建立起一个输出流,通向由这些节点所构成的流水线 Pipeline。先看 `locateFollowingBlock()`:

```
[DFSOutputStream.DataStreamer.run() > nextBlockOutputStream() > locateFollowingBlock()]
```

```
locateFollowingBlock(long start, DatanodeInfo[] excludedNodes)
```

```

> retries = dfsClient.getConf().nBlockWriteLocateFollowingRetry
> while (true) {
>+ localstart = Time.now()
>+ while (true) {
>++ try {
>++ return dfsClient.namenode.addBlock(src, dfsClient.clientName, block,
                                     excludedNodes, fileId, favoredNodes)
//通过 RPC 向 NameNode 要求添加数据块
//希望避免 excludedNodes 中那些节点,最好能安排在 favoredNodes 中那些节点上
>++ } catch (RemoteException e) {
>+++ ...
>++ }
>+ } //end while
> } //end while

```

这里我们看到了对 NameNode 的 RPC 调用 `addBlock()`,向 NameNode 要求向目标文件(当然是 HDFS 文件)中添加一个数据块,并请告知应该把数据块发送给哪些 DataNode。

对此,在 NameNode 这一边照例还是先到 NameNodeRpcServer:

```

NameNodeRpcServer.addBlock(String src, String clientName, ExtendedBlock previous,
                           DatanodeInfo[] excludedNodes, long fileId, String[] favoredNodes)
> if (excludedNodes != null) { //这些节点应于排除
>+ excludedNodesSet = new HashSet<Node>(excludedNodes.length)
>+ for (Node node : excludedNodes) excludedNodesSet.add(node)
> }
> favoredNodesList = (favoredNodes == null)?null : Arrays.asList(favoredNodes)
//这些是偏好的节点
> LocatedBlock locatedBlock = namesystem.getAdditionalBlock(src, fileId,

```

```

        clientName, previous, excludedNodesSet, favoredNodesList)
    == FSNamesystem.getAdditionalBlock(src, fileId, clientName, previous,...)
> return locatedBlock

```

对于本书的读者,这已是熟知的套路了,NameNodeRpcServer.addBlock()还是通过调用FSNamesystem提供的方法来完成操作:

```

[NameNodeRpcServer.addBlock() > FSNamesystem.getAdditionalBlock()]

```

```

FSNamesystem.getAdditionalBlock(String src, long fileId, String clientName,
    ExtendedBlock previous, Set<Node> excludedNodes, List<String> favoredNodes)
> // Part I. Analyze the state of the file with respect to the input data.
> checkOperation(OperationCategory.READ)
> byte[][] pathComponents = FSDirectory.getPathComponentsForReservedPath(src)
> src = resolvePath(src, pathComponents) //目标文件的文件名
> LocatedBlock[] onRetryBlock = new LocatedBlock[1]
> FileState fileState = analyzeFileState(src, fileId, clientName, previous, onRetryBlock)
    //在那里会 checkLease(),此地不展开,留给读者自行分析
> INodeFile pendingFile = fileState.inode
> src = fileState.path
> if (onRetryBlock[0] != null && onRetryBlock[0].getLocations().length > 0) {
>+ // This is a retry. Just return the last block if having locations.
>+ return onRetryBlock[0]
> }
> if (pendingFile.getBlocks().length >= maxBlocksPerFile)
    IOException("File has reached the limit on maximum number of" ...)
> blockSize = pendingFile.getPreferredBlockSize()
> clientMachine = pendingFile.getFileUnderConstructionFeature().getClientMachine()
> clientNode = blockManager.getDatanodeManager().getDatanodeByHost(clientMachine)
    //这是 Client 所在的节点,如果不在 DataNode 上就返回 null
> replication = pendingFile.getFileReplication() //目标文件所设定的复份个数
> storagePolicyID = pendingFile.getStoragePolicyID()
> if (clientNode == null) clientNode = getClientNode(clientMachine)
> // choose targets for the new block to be allocated.
> DatanodeStorageInfo targets[] = getBlockManager().chooseTarget4NewBlock(src,
    replication, clientNode, excludedNodes, blockSize, favoredNodes, storagePolicyID)
    //由 BlockManager 为新数据块选择一组存储地点
> // Part II. Allocate a new block, add it to the INode and the BlocksMap.
> checkOperation(OperationCategory.WRITE)
> waitForLoadingFSImage()
> // Run the full analysis again, since things could have changed

```

```

while chooseTarget() was executing.
> LocatedBlock[] onRetryBlock = new LocatedBlock[1]
> FileState fileState = analyzeFileState(src, fileId, clientName, previous, onRetryBlock)
    // 再次调用这个函数
    // 因为目标文件的状态在 chooseTarget4NewBlock()期间可能已经发生变化
> INodeFile pendingFile = fileState.inode
> src = fileState.path
> if (onRetryBlock[0] != null) {
>+ if (onRetryBlock[0].getLocations().length > 0) {
>++ // This is a retry. Just return the last block if having locations.
>++ return onRetryBlock[0]
>+ } else {
>++ // add new chosen targets to already allocated block and return
>++ BlockInfo lastBlockInFile = pendingFile.getLastBlock()
>++ ((BlockInfoUnderConstruction) lastBlockInFile).setExpectedLocations(targets)
>++ offset = pendingFile.computeFileSize()
>++ return makeLocatedBlock(lastBlockInFile, targets, offset)
>+ }
> } //end if (onRetryBlock[0] != null)
> // commit the last block and complete it if it has minimum replicas
> commitOrCompleteLastBlock(pendingFile, ExtendedBlock.getLocalBlock(previous))
> // allocate new block, record block locations in INode.
> newBlock = createNewBlock()
> INodesInPath inodesInPath = INodesInPath.fromINode(pendingFile)
> saveAllocatedBlock(src, inodesInPath, newBlock, targets)
> persistNewBlock(src, pendingFile)
> offset = pendingFile.computeFileSize()
> getEditLog().logSync() // 同步 EditLog
> return makeLocatedBlock(newBlock, targets, offset)
    // 为数组 targets 中的节点创建一个 LocatedBlock 对象,并返回给申请者

```

摘要中已加了一些注释,这里不再详述,留给读者自己阅读分析。这个函数的返回值是个 LocatedBlock 对象,其核心是个 DatanodeInfo 对象数组,这就是下一个块的诸多复份的存放地点。NameNode 上的 PB 层和 RPC 层会把这个 LocatedBlock 对象搭载在响应报文上发回 DataNode,我们在前面见过类似的细节,这里就不再赘述了。

回到 DataNode 这边的 nextBlockOutputStream()中,locateFollowingBlock()的返回值就是来自 NameNode 的 LocatedBlock 对象,这被赋值给变量 lb,本质上就是一个用来存放下一个块(各个复份)的节点清单。有了这个清单之后,就可以通过 createBlockOutputStream()构建通向这些节点的流水线了。

```

[DataStreamer.run()>nextBlockOutputStream()>createBlockOutputStream()]

createBlockOutputStream(DatanodeInfo[] nodes,
                        StorageType[] nodeStorageTypes, long newGS, boolean recoveryFlag)
> persistBlocks.set(true) //persist blocks on namenode on next flush
> while (true) {
>+ s = createSocketForPipeline(nodes[0], nodes.length, dfsClient) //建立 Socket
>+ OutputStream unbufOut = NetUtils.getOutputStream(s, writeTimeout)
>+ InputStream unbufIn = NetUtils.getInputStream(s)
>+ IOStreamPair saslStreams = dfsClient.saslClient.socketSend(s, unbufOut, unbufIn,
                        dfsClient, accessToken, nodes[0])//在原始 Socket 上加 SASL 加密层
>+ unbufOut = saslStreams.out
>+ unbufIn = saslStreams.in
>+ out = new DataOutputStream(new BufferedOutputStream(unbufOut,
                        HdfsConstants.SMALL_BUFFER_SIZE)) //加缓冲存储
>+ blockReplyStream = new DataInputStream(unbufIn)
>+ BlockConstructionStage bcs = recoveryFlag?stage.getRecoveryStage(): stage
>+ ExtendedBlock blockCopy = new ExtendedBlock(block)
                        //注意这个 block 只是关于 Block 的说明,例如块号等等,此时尚无内容。
>+ new Sender(out).writeBlock(blockCopy, nodeStorageTypes[0], accessToken,
                        dfsClient.clientName, nodes, nodeStorageTypes, null, bcs,
                        nodes.length, block.getNumBytes(), bytesSent, newGS,
                        checksumWriteBlock, cachingStrategy.get(), isLazyPersistFile)
                        == Sender.writeBlock(final ExtendedBlock blk,...)
>+> ...
>+> send(out, Op.WRITE_BLOCK, proto.build())
>+ BlockOpResponseProto resp = BlockOpResponseProto.parseFrom(
                        PBHelper.vintPrefixed(blockReplyStream)) //获取下游的响应
>+ pipelineStatus = resp.getStatus()
>+ firstBadLink = resp.getFirstBadLink()
>+ ...
>+ return result
> } //end while,如果不发生异常就不会进入第二轮循环

```

这里之所以有个 while 循环,是为了出错以后的重试,源代码中这些操作都放在一个 try{} catch(){}finally{} 结构中,只是在做摘要时把它简化了。如果不发生异常,那么程序在末尾的那个 return 语句就会返回,而不会进入第二轮循环。

参数 nodes 就是来自 NameNode 的那个 DatanodeInfo 数组,这些数组元素所代表的那些 DataNode,按其出现在数组中的次序,将构成一个数据块传输流水线(Pipeline),数组中的第一个元素即 nodes[0]就是这个流水线的头。需要发送什么 Packet 时,DataStreamer 只要把它发送给这流水线中的第一个节点,后面就由这些节点自己逐站转发了。

这里在创建了基于 Socket 的输出流之后在此基础上临时创建了一个 Sender,并调用其 writeBlock() 向此 Pipeline 中的第一个节点发送了一个 WRITE_BLOCK 指令,里面包括了目标数据块的 ExtendedBlock 对象(数据结构)、Pipeline 的构成即数组 nodes 等信息。

这个写块指令实质上是个建立流水线的指令,真正的数据发送还没有开始。至于对方节点在接收到 WRITE_BLOCK 指令后的反应,则我们在前一章中已经看到过了。

注意这里向下游方向发出的请求是 WRITE_BLOCK。但是这个请求只在 DataStreamer 的流水线构筑阶段 stage 为 PIPELINE_SETUP_CREATE 或 PIPELINE_SETUP_APPEND 时才发送一次,发送之后就进入了 DATA_STREAMING 阶段,以后就不会再发送这个请求了,除非 App 又调用了一次 DFSClient.create()或 DFSClient.append()。但是发送了 WRITE_BLOCK 请求并不意味着已经有了这个块的数据而马上可以开始发送,这取决于 App 是否已经向这个输出流写入了足够多的数据。如果一时还没有写入数据,那么 DataStreamer 的 dataQueue 中就没有 Packet 可供发送,如果时间长了就得生成一个 HeartbeatPacket 充数。而在 DataNode 那一头,则在收到 WRITE_BLOCK 之后并不要求马上就有数据到来,只是不能长时间杳无音信。如果到来的是 HeartbeatPacket 而不是数据 Packet,就会因为里面没有数据而予以丢弃。

那什么时候才有数据? 这取决于 App,得要 App 往文件中写,才有数据需要发送。

DFSOutputStream 本身并未提供 write() 操作,但 DFSOutputStream 是对抽象类 FSOutputSummer 的扩充,而后者定义了 write(),所以 DFSOutputStream 继承了后者的这个 write(),我们就从这里开始:

```

FSOutputSummer.write(byte b[], int off, int len)
> for (int n = 0; n < len; n += write1(b, off + n, len - n)) { //empty }
    == FSOutputSummer.write1(byte b[], int off, int len)
>> if(count == 0 && len >= buf.length) { //buf 中无数据,而要写入的数据则达到 buf 的容量
>>+ length = buf.length
>>+ writeChecksumChunks(b, off, length) //把 buf 中的数据作为若干个 Chunk 发送
>>+ return length //从 write1()返回,进入下一轮 for 循环
>> }
>> bytesToCopy = buf.length - count //否则就在缓冲区 buf 中积累数据
>> bytesToCopy = (len < bytesToCopy)?len : bytesToCopy
>> System.arraycopy(b, off, buf, count, bytesToCopy) //把一部分数据拷贝到 buf 中
>> count += bytesToCopy //buf 中的数据增加了
>> if (count == buf.length) { //local buffer is full,如果缓冲区满了
>>+ flushBuffer() //那就冲刷出去
>>+ flushBuffer(false, true) == flushBuffer(boolean keep, boolean flushPartial)
>>+>> bufLen = count
>>+>> partialLen = bufLen % sum.getBytesPerChecksum()
    //sum 是个 DataChecksum 对象,getBytesPerChecksum()返回每次 CRC 校验的长度
>>+>> lenToFlush = flushPartial?bufLen : bufLen - partialLen
    //本次冲刷的长度,应该是 CRC 校验长度即 Chunk 长度的整数倍

```

```

>>+>> if (lenToFlush != 0) { //lenToFlush 为 0 表示还不到一个 Chunk,那就继续积累
>>+>>+ writeChecksumChunks(buf, 0, lenToFlush) //发送若干个 Chunk 的数据
>>+>>+ if (!flushPartial || keep) {
>>+>>+ } else count = 0
>>+>> }
>>+>> return count - (bufLen - lenToFlush) //从 flushBuffer()返回
>> } //end if (count == buf.length)
>> return bytesToCopy //返回到 write()的 for 循环中

```

在缓冲区中写满了数据之后,就以 Chunk,即以实施 CRC 校验的长度为单位通过 writeChecksumChunks()向外写出:

```
[FSOutputSummer.write() > writel() > writeChecksumChunks()]
```

```
FSOutputSummer.writeChecksumChunks()
```

```

> sum.calculateChunkedSums(b, off, len, checksum, 0) //CRC 校验计算
> for (int i = 0; i < len; i += sum.getBytesPerChecksum()) { //每轮循环一个 Chunk
>+ chunkLen = Math.min(sum.getBytesPerChecksum(), len - i)
>+ ckOffset = i / sum.getBytesPerChecksum() * getChecksumSize()
>+ writeChunk(b, off + i, chunkLen, checksum, ckOffset, getChecksumSize())
    == DFSOutputStream.writeChunk(byte[] b, int offset, int len,
                                   byte[] checksum, int ckoff, int cklen)
>+> if (currentPacket == null) { //如果还没有开始构建 Packet 内容
>+>+ currentPacket = createPacket(packetSize, chunksPerPacket,
                                   bytesCurBlock, currentSegno++) //创建一个 Packet
>+> }
>+> currentPacket.writeChecksum(checksum, ckoff, cklen) //写入 Packet 的 CRC 校验码部分
>+> currentPacket.writeData(b, offset, len) //写入 Packet 的数据部分
>+> currentPacket.numChunks++ //Packet 中的 Chunk 数量加 1
>+> bytesCurBlock += len
>+> if (currentPacket.numChunks == currentPacket.maxChunks || bytesCurBlock == blockSize){
    //如果已经积累了够一个 Packet 的数据:
>+>+ waitAndQueueCurrentPacket() //等待发送队列中有空位时,挂入 Packet。
>+>+> while (!closed && dataQueue.size() + ackQueue.size()
    > dfsClient.getConf().writeMaxPackets) {
>+>+>+ dataQueue.wait() //等待 dataQueue 中有空位
>+>+> } //end while
>+>+> queueCurrentPacket() //将 Packet 挂入队列
>+>+>> dataQueue.addLast(currentPacket) //挂在 dataQueue 的末尾
>+>+>> lastQueuedSegno = currentPacket.segno
>+>+>> currentPacket = null // currentPacket 已经离开缓冲区

```



```

>+>+>> dataQueue.notifyAll() //唤醒 DataStreamer 线程
        // -- waitAndQueueCurrentPacket() 结束
>+>+ if (appendChunk && bytesCurBlock % bytesPerChecksum == 0) {
>+>+ appendChunk = false
>+>+ resetChecksumBufSize()
>+>+ }
>+>+ if (!appendChunk) {
>+>+ psize = Math.min((int)(blockSize - bytesCurBlock), dfsClient.getConf().writePacketSize)
>+>+ computePacketChunkSize(psize, bytesPerChecksum)
>+>+ }
>+>+ if (bytesCurBlock == blockSize) {
        //如果到了 Block 的末尾,就要发一个空的 Packet 以示结束
>+>+ currentPacket = createPacket(0, 0, bytesCurBlock, currentSeqno++) //创建空的 Packet
>+>+ currentPacket.lastPacketInBlock = true
>+>+ currentPacket.syncBlock = shouldSyncBlock
>+>+ waitAndQueueCurrentPacket() //将表示 Block 结束的空 Packet 挂入发送队列
>+>+ bytesCurBlock = 0
>+>+ lastFlushOffset = 0
>+>+ } //end if (bytesCurBlock == blockSize)
>+> } //end if (currentPacket.numChunks == currentPacket.maxChunks...)
> } //end for

```

这样,作为对 FSOutputStream 的扩充,DFSOutputStream 将 App 写入其缓冲区的数据每积累到一定程度就构成一个 Packet,并将其挂入它的发送队列 dataQueue。如果到了一个块的长度就增加一个空 Packet 以示块的结束。注意,DFSOutputStream 本身并非一个线程,这些程序都是在 App 的上下文中执行,直到将一个个的 Packet 挂入发送队列。但是 DFSOutputStream 确实有个 Packet 发送线程,这就是上述的 DataStreamer 线程 streamer。这个线程不断从发送队列 dataQueue 取出 Packet 加以发送,至于发送的目的地则如前所说来自 NameNode 的指定。

建立了这个通往 Pipeline 的输出流,程序逐层返回到前面 DataStreamer.run()的代码中,后面调用 one.writeTo(blockStream),即 Packet.writeTo(blockStream),那就是把一个 Packet 写入这个输出流 blockStream;然后再调用 blockStream.flush(),就把这个 Packet 发送出去了。

当然,一个数据块会被分成很多 Packet 发送(须知一个典型数据块的大小是 64MB),所以会在 DataStreamer.run()的那个 while 循环中循环很多次。每发送一个 Packet,就把这个 Packet 从 dataQueue 队列转移到 askQueue 中,而每当对方做出对接收到某个 Packet 的响应,这一边的 ResponseProcessor 线程就从 ackQueue 中把这个 Packet 去掉。这样,ackQueue 队列中还有几个 Packet,就是已发送而尚未得到对方回应的那些 Packet。之所以把已经发送的 Packet 留在 ackQueue 队列中,一方面固然是要有个类似于销号的机制,但更重要的显然是考虑到失败后重发的问题,不过在这个问题上我们就不深入下去了。

发送之后,下游方向将会有响应信息发回,ResponseProcessor 线程是在前面的 initDataStreaming()中创建的,这个线程的代码摘要留给读者自己阅读:

```
class ResponseProcessor extends Daemon {}
] DatanodeInfo[] targets
] run()
    > PipelineAck ack = new PipelineAck() //创建一个空白的 PipelineAck 对象
    > while (!responderClosed && dfsClient.clientRunning && !isLastPacketInBlock){
    >+ ack.readFields(blockReplyStream)//从下游读入一个 PipelineAck 到 ack 中
    >+ seqno = ack.getSeqno()
    >+ // processes response status from datanodes.
    >+ for (int i = ack.getNumOfReplies() - 1; i >= 0 && dfsClient.clientRunning; i--) {
    >++ Status reply = ack.getReply(i)
    >++ if (PipelineAck.isRestartOOBStatus(reply) && shouldWaitForRestart(i)) {
    //若其中有 DataNode 重启
    >+++ restartDeadline = dfsClient.getConf().datanodeRestartTimeout + Time.now()
    >+++ setRestartingNodeIndex(i)
    >+++ IOException("A datanode is restarting: " + targets[i])
    >++ }
    >++ if (reply != SUCCESS) IOException("Bad response " + reply +
    " for block " + block + " from datanode " + targets[i])
    >+ } //end for
    >+ if (seqno == Packet.HEART_BEAT_SEQNO) continue //a heartbeat ack
    >+ // a success ack for a data packet
    >+ Packet one = ackQueue.getFirst() //等待确认的 Packet 都在 ackQueue 中
    >+ if (one.seqno != seqno) IOException("ResponseProcessor: Expecting seqno " + ...)
    >+ isLastPacketInBlock = one.lastPacketInBlock
    >+ if (DFSClientFaultInjector.get().failPacket() && isLastPacketInBlock)
    IOException("Failing the last packet for testing.")
    >+ block.setNumBytes(one.getLastByteOffsetBlock()) //update bytesAked
    >+ lastAkedSeqno = seqno
    >+ ackQueue.removeFirst() //已经得到下游确认,无须重发,可以从 ackQueue 中撤除
    >+ dataQueue.notifyAll()
    >+ one.releaseBuffer(byteArrayManager) //这个 Packet 已发送成功
    > } //end while
```

至于对方怎样接收和处理发送过去的 Packet,我们已经在前一章中看到过了。不过前一章中并未说明对方怎样做出回应,这也留给读者自己阅读了。

ResponseProcessor 只管根据下游的应答从 ackQueue 队列中“销号”,而不管未获销号的那些 Packet 的重发。如果发送过程中有错,或有目标节点重启,使已经发送并移入 ackQueue 的 Packet 没有得到应答确认,就把 ackQueue 队列中的 Packet 移回 dataQueue,以供重发。见

DataStream.run()中对 processDatanodeError()的调用。

注意 DataStreamer 线程只管一个数据块的发送,实际上每个块的存储地点不同,从而数据传送的流水线也不同。当一个块的所有 Packet 都发送出去之后,要发送一个空的 Packet 以示数据块的结束,到所有的 Packet 都得到回应之后,这个线程就已完成使命,应该退出了。再往下,如果还要继续写这个 HDFS 文件,就要另起一个数据块,那就要另外创建一个 DataStreamer 线程,另外再向 NameNode 申请 addBlock(),另外建立一个流水线了。

最后,写完了文件内容之后,要通过 close()加以关闭才算完成。

```
DFSOutputStream.close()
> flushBuffer();           // flush from all upper layers
> if (currentPacket != null) waitAndQueueCurrentPacket()
> if (bytesCurBlock != 0) {
>+ // send an empty packet to mark the end of the block
>+ currentPacket = createPacket(0, 0, bytesCurBlock, currentSeqno++) //创建一个空 Packet
>+ currentPacket.lastPacketInBlock = true
>+ currentPacket.syncBlock = shouldSyncBlock
> }
> flushInternal();          // flush all data to Datanodes
>> dfsClient.checkOpen()
>> checkClosed()
>> queueCurrentPacket()
>> toWaitFor = lastQueuedSeqno
>> waitForAkedSeqno(toWaitFor)
> // get last block before destroying the streamer
> ExtendedBlock lastBlock = streamer.getBlock()
> closeThreads(false)
> completeFile(lastBlock)
>> retries = dfsClient.getConf().nBlockWriteLocateFollowingRetry
>> while (!fileComplete) {
>>+ fileComplete = dfsClient.namenode.complete(src, dfsClient.clientName, last, fileId)
//通过 RPC 向 NameNode 发出关闭文件的通知
>>+ if (!fileComplete) {
>>++ if (retries == 0) IOException("Unable to close file because the last block
does not have enough number of replicas."
>>++ retries --
>>++ Thread.sleep(localTimeout)
>>++ localTimeout * = 2
>>+ }
>> }
> dfsClient.endFileLease(fileId) //放弃租约
```

而 NameNode 这一边,则通过 FSNamesystem.complete()完成文件的关闭:

```

NameNodeRpcServer.complete(String src, String clientName, ExtendedBlock last, long fileId)
> return namesystem.completeFile(src, clientName, last, fileId)
== FSNamesystem.complete(String src, String clientName, ExtendedBlock last, long fileId)
>> checkBlock(last)
>> checkOperation(OperationCategory.WRITE)
>> byte[][] pathComponents = FSDirectory.getPathComponentsForReservedPath(src)
>> waitForLoadingFSImage()
>> checkNameNodeSafeMode("Cannot complete file " + src)
>> src = resolvePath(src, pathComponents)
>> success = completeFileInternal(src, holder, ExtendedBlock.getLocalBlock(last), fileId)
>>> if (fileId == INodeId.GRANDFATHER_INODE_ID) {
>>>+ INodesInPath iip = dir.getLastINodeInPath(src)
>>>+ inode = iip.getINode(0)
>>> } else {
>>>+ inode = dir.getINode(fileId)
>>>+ if (inode != null) src = inode.getFullPathName()
>>> }
>>> INodeFile pendingFile = checkLease(src, holder, inode, fileId)
>>> if (!checkFileProgress(pendingFile, false)) return false
>>> // commit the last block and complete it if it has minimum replicas
>>> commitOrCompleteLastBlock(pendingFile, last)
>>> if (!checkFileProgress(pendingFile, true)) return false
>>> finalizeINodeFileUnderConstruction(src, pendingFile, Snapshot.CURRENT_STATE_ID)
// 目标文件的 INodeFile 节点前已转成 UC 状态,现在转回正常,并记入 EditLog
>>> return true
>> getEditLog().logSync()
>> return success

```

可见,作为用户与 NameNode 之间的中介,DFSClient 其实也相当于 HDFS 的上层,可以看成是 HDFS 的一部分,与 NameNode、DataNode 合在一起就成为 Hadoop 的文件系统。再结合 DFSOutputStream 和 DFSInputStream,那相当于通向已打开文件的输出流和输入流,就为用户提供了一个类似于 POSIX 文件系统那样的 API。但毕竟还是有明显的不同,最显著的是 HDFS 文件不支持随机写入;另外,为了读而打开 HDFS 文件要用 open(),为了写而打开 HDFS 文件却要用 create()或 append(),这也是有点特殊的安排。

15.8 实例

明白了 HDFS 的文件操作之后,我们不妨再通过几个实例看一下 Client 对文件系统的访问。前面我们曾看过命令 Cat 的执行,那只涉及读 HDFS 文件,现在再看另外一个 Shell 命令

appendToFile,使用时的命令行大致上是这样:“hdfs dfs -appendToFile src1 src2 dst”。这里 src1 和 src2 是本地即宿主文件系统中的两个文件,dst 是 HDFS 文件系统中的文件,这条命令依次把 src1 和 src2 的内容粘贴到 dst 的末尾,这很像本来意义上的 cat。这条命令使 FsShell 创建一个定义于 CopyCommands 类内部的 AppendToFile 类对象,并加以执行。如果读者已经忘记 processArguments()是怎么回事,可以回到前面看一下 Cat 命令的执行过程。

```
class CopyCommands.AppendToFile extends CommandWithDestination {} //"- appendToFile"
] String NAME = "appendToFile"
] String USAGE = "<localsrc> ... <dst>"
] String DESCRIPTION = "Appends the contents of all the given local files to the " + ...
] processArguments(LinkedList<PathData> args)
    > if (!dst.exists) dst.fs.create(dst.path, false).close()
        //如果 HDFS 文件 dst 原来没有,就 create()一下并马上 close(),这样就一定存在了
    > FSDataOutputStream fos = dst.fs.append(dst.path) //打开目标文件,在其尾部添加
        //因为 dst 一定存在,所以可以统一用 append(),返回的输出流为 fos
    > if (readStdin) { //如果粘贴的内容不是来自文件而是来自键盘
    >+ if (args.size() == 0) IOUtils.copyBytes(System.in, fos, DEFAULT_IO_LENGTH)
    >+ else IOException("stdin (-) must be the sole input argument when present")
    > }
    > // Read in each input file and write to the target.
    > for (PathData source : args) { //对于每个输入文件
    >+ is = new FileInputStream(source.toFile()) //打开这个文件,成为输入流 is
    >+ IOUtils.copyBytes(is, fos, DEFAULT_IO_LENGTH)
    >+ IOUtils.closeStream(is) //关闭输入流,即源文件
    >+ is = null
    > }
    > if (is != null) IOUtils.closeStream(is)
    > if (fos != null) IOUtils.closeStream(fos) //关闭输出流,即目标文件
```

这里的 copyBytes()从输入流 is 拷贝内容到作为输出流的 HDFS 文件 dst:

```
IOUtils.copyBytes(InputStream in, OutputStream out, int buffSize)
> PrintStream ps = out instanceof PrintStream?(PrintStream)out : null
        //把输出流当作打印流 PrintStream,这是为了检错,不用管它
> byte buf[] = new byte[buffSize]
> int bytesRead = in.read(buf) //从输入流读入
> while (bytesRead >= 0) {
>+ out.write(buf, 0, bytesRead)
        == DFSDataOutputStream.write(buf, 0, bytesRead)
>+ if ((ps != null) && ps.checkError()) IOException("Unable to write to output stream.")
>+ bytesRead = in.read(buf)
> }
```

可见,除前面打开文件时有所不同之外,后面的 `copyBytes()` 中看不出与针对一般文件的操作有什么不同。

但请注意,上面在关闭文件时用的是 `IOUtils.closeStream()`,而不是 `DFSClient.close()`,前者是对个别文件,即输入或输出流的关闭;而后者则是对整个 `DFSClient`,即通往 HDFS 的整个中介的关闭。

另有个很有意思的实例是 HDFS 文件系统的负载均衡。Hadoop 集群经过一段较长时间的运行之后,各 `DataNode` 节点的负载可能会很不均匀,就是有些节点上存放了太多的块文件,而有些节点上却有大片空间闲置。这会显著影响 HDFS 的性能。或者,设想一个集群原来有 1000 个 `DataNode` 节点,现在增加了 500 个节点,这新增的 500 个 `DataNode` 一开始完全没有负载。所以,Hadoop 提供了一个工具 `balancer`,用来在节点间搬运数据块复份,使负载变得均匀。这个软件是单独作为一个 App 运行的,输入命令行“`hdfs start balancer`”就可以看到使用提示。这个软件的主类是 `Balancer`,限于本书篇幅这里就不作分析和讲解了,有兴趣的读者不妨自己分析一下,于加深对 HDFS 的理解一定大有好处。

第16章

Hadoop 的容错机制

16.1 容错与高可用

人的健康与否,就看其承受和克服疾病、伤痛、老化和不利环境的能力,这跟力气大不大、跑得快不快、智商高不高没有什么关系。同样,一个系统之是否强健(Robust),就看其承受和克服故障的能力。这是因为,别的因素如元器件的老化和失效、人为的误操作(例如不经正常的关机程序 shutdown 就关电源)乃至应用软件本身的问题,最终都表现为故障。所以,“容错(Fault Tolerant)”是计算机系统的一个重要问题,对于大规模的计算机集群就更是如此。“容错”这个词,一方面是说系统能承受故障,即使有了故障也不会对功能的发挥带来太大的影响,特别是不能有“跳崖”式的功能下降;另一方面是说系统能克服和消除故障带来的后果,从不利的后果中恢复过来。不过,能不能恢复过来是一回事;需要多长时间和什么样的措施才能恢复过来,在此过程中系统的服务是否中断,又是一回事。比方说,如果我们不经 shutdown 就关了电源,就可能给文件系统带来损害,下次开机时就可能要执行 fsck。多数情况下 fsck 都能修复文件系统的故障,能从故障中恢复过来,所以也可以说在某种程度上是“容错”的,但是执行 fsck 的时间却可能有点长,在这段时间中系统实际上是不可用的。另外,对于具体 App 的运行而言,系统一出故障,这一次特定的运行可能就失败了;至于下一次重新执行,有时候是可以接受的,有时候却未必。

会引起系统在一个时间段中不可用的原因还不仅仅是故障,例如系统软件的升级、电路板卡的更换等例行或偶发的维护都有可能需要让机器暂时退出服务。对于像 Hadoop 这样的集群,以 HDFS 为例,让个别 DataNode 暂时退出服务倒问题不大,但要是让 NameNode 暂时退出服务,如果没有特别的措施的话,那么整个集群的 HDFS 文件系统就不能动了,这意味着整个集群在这个时间段中就不可用了。

这样的情况,即系统可能在某些时间段中不可用,对于一般的个人计算机应用可能无关紧要,但是对有些系统的应用却至关重要。在这方面要求比较高、比较严格的系统,要求即使出了故障也要让用户基本无感,就是让人觉得性能上似乎有些下降甚或迟滞,但还是在正常的性能波动范围之内,能做到让用户毫无感觉那当然更好。或者,对于多用户、多任务的系统,即使有那么几个用户或任务明显感到了故障的影响,别的大多数的用户和任务却应该基本不受影响。

所以,“高可用(High Availability, HA)”是更高的容错要求。作为一个系统,一个平台(Platform),“高可用”也就是“高可靠”,它不会忽然间就用不上了。不过 HA 与一般意义上的

容错之间也并非那么黑白分明,系统的容错能力提高一些,可用性也自然就提高一些;另一方面,对于有些系统,要让用户承认你这是“容错”,事实上就必须是某种程度的“高可用”才行。所以“容错”这个词的意思有广义和狭义之分,有时候它直接就是“高可用”的意思。

可想而知,高可用,即 HA,也包括一般的容错,是一定要通过设备、功能、信息甚至时间,总而言之是资源的冗余才能达到的。进一步,也是可想而知,HA 的关键在于某些功能或信息集中的单点。以 Hadoop 为例,其资源管理节点 RM 和文件系统的 NameNode,即 NN,就是这样的单点,那就是 HA 的关键所在。

就 Hadoop 而言,一般意义上的容错当然是必须的。比方说,整个系统突然断了电,这时候文件系统 HDFS 可能就不一致了,如果恢复不过来岂不糟糕?这时候 Checkpoint 和 EditLog 这套机制就起作用了。集群中有那么多的数据节点(DataNode),难免常有节点损坏,而文件的内容都分布存储在许多节点上,如果每个数据块都只存一份的话,只要其中有一个节点损坏就可以损害一大批文件,理论上甚至可能是所有文件。而文件内容 Block 的多复份存储,即按 Replica 存储,就是为了解决这个问题。再如 YARN 的设计把每个具体应用的管理 AM 分布出去而不再集中在 RM 节点上,一方面当然是为了减轻 RM 节点的负担,另一方面也有助于减轻单点故障的影响,有助于容错。所以,Hadoop 固有的设计中其实已经采取了许多容错的措施。而更高要求的 HA,则是作为选项提供的;而且 HA 的选用与否并非整体上的有或没有,而是可以针对 YARN 的 RM 和 HDFS 的 NN 分别采用。进一步,在“联邦(Federation)”模式下,集群中可以有多个 NameNode,即多个 HDFS 文件系统,那就可以针对具体的 NameNode 分别决定是否采用 HA 选项。Hadoop 代码中凡属于 HA 选项的程序都放在条件语句中。

在 YARN 的配置文件 yarn-default.xml 中,有个属性“yarn.resourcemanager.ha.enabled”,把这个属性的值设置成 true,就表示要为 YARN,实际上就是 RM 节点,配上 HA 选项。在 Hadoop 的代码中,字符串 RM_HA_ENABLED 就代表着这个选项。这个选项的默认值是 false,即不采用额外的 HA 措施,而只是采用一般的容错。Hadoop 的源码文件“.../yarn/conf/HAUtil.java”中有个函数 isHAEnabled(),用来从配置中获取这个属性的值:

```
public static boolean isHAEnabled(Configuration conf) {    //用于 YARN
    return conf.getBoolean(YarnConfiguration.RM_HA_ENABLED,
                           YarnConfiguration.DEFAULT_RM_HA_ENABLED);
}
```

代码中的 RM_HA_ENABLED 就是属性的名称“yarn.resourcemanager.ha.enabled”,而 DEFAULT_RM_HA_ENABLED 则定义为“false”。如果在配置文件中找不到那个属性,就默认为 false。

注意,这是在“.../yarn/conf”目录下的 HAUtil.java 中,在 hdfs 目录下也有一个同名的文件。HAUtil 是一个 Java 类,但是实际上只是一组工具性的小函数,Hadoop 系统中没有一个具体的对象是属于这个类的,要调用这个函数就得写 HAUtil.isHAEnabled()。

对于 HDFS 还有些不同,因为现在的 Hadoop 支持联邦模式,集群中可能不止一个文件系统,从而不止一个 NameNode,但是不同文件系统的重要性可能不一样,所以 HA 选项是针对具体 NameNode 的,更确切地说是针对一套具体的查名服务(nameservice)的。源码文件

“.../hdfs/HAUtil.java”中也有一个 isHAEnabled():

```
public static boolean isHAEnabled(Configuration conf, String nsId) {    //用于 HDFS
    Map<String, Map<String, InetSocketAddress>>> addresses =
        DFSUtil.getHaNnRpcAddresses(conf);
    if (addresses == null) return false;    //没有配置专用于 HA 的 RPC 地址,就是不采用 HA
    Map<String, InetSocketAddress> nnMap = addresses.get(nsId);    //具体 nameservice 的地址
    return nnMap != null && nnMap.size() > 1;    //为这个具体 nameservice 配置了 RPC 地址
}
```

与前面 YARN 的那个 isHAEnabled()相比,这里多了一个参数 nsId,这是 NameServiceId 的意思,所以是针对具体 nameservice 的,一般也可以理解为是针对具体 NameNode 的(但不排除一个 nameservice 有多个 NameNode)。对于 HA 的支持离不开冗余,具体就是节点的热备份,这就需要有 RPC 通信,互相就要有对方的地址,所以这就可以用作判定的条件。

配置文件 hdfs-default.xml,实际使用时就成为 hdfs-site.xml,里面有一个属性“dfs.namenode.rpc-address”,在这个属性中可以列举集群中所有 NameNode 的 RPC 地址,地址的形式为 URI 节点名加端口号。当 NameNode 不止一个的时候,就要把 nsId 编码在节点名中,如 dfs.namenode.rpc-address.ns1。此外配置文件中还有个属性“dfs.nameservices”,下面列出了系统中所有 nameservice 的名称,即 Id。另一个以“dfs.ha.namenodes”为前缀的属性则列举提供某种特定 nameservice 的 NameNode 节点。而 getHaNnRpcAddresses(),则首先要从配置文件中找到所有的 nameservice,再找到与各项 nameservice 相关的 RPC 地址,所以 getHaNnRpcAddresses()这个函数其实并不简单。注意配置文件 hdfs-site.xml 是局部于具体节点,而不是全局的,集群中的每个机器节点上都有自己的这个配置文件。对于 RM 的配置也是一样。Hadoop 的源码包中有个目录“hdfs-project/hadoop-hdfs/src/site”,下面有许多 .vm 文件,这些文件的集合实际上构成 HDFS 的用户手册,其中的 Federation.appt.vm、HDFSHighAvailabilityWithNFS.appt.vm 等文件中有关于如何设置这些属性的说明,读者可以参阅。有兴趣或需要的读者更可深入 getHaNnRpcAddresses()的代码,以求彻底搞个明白。

显然,这个 isHAEnabled()的执行开销是不小的,所以 Hadoop 的代码中都是先调用一下这个函数,将结果赋给一个变量 haEnabled,以后在条件语句中就只是使用这个变量了。注意,是否使用 HA 是针对具体 NameNode 和 FSNamesystem 的,会因不同的 NameNode 和 FSNamesystem 而异,所以 NameNode 和 FSNamesystem 对象内部都有自己的 haEnabled。

鉴于一般的容错措施已经渗透在 Hadoop 的设计之中,所以本章主要讲述作为选项提供的、要求更高更严格的 HA 机制。不过在此之前我们不妨自底向上地先回顾和分析一下 Hadoop 基本的容错措施。

首先,作为一个集群,网络通信的可靠性当然是个基础。在这方面,集群内部肯定是局域网的连接,流量无须进入公网,节点互连的可靠性还是比较高的。再说,所采用的通信协议默认为 TCP 而不是 UDP,如果发生丢包就会重发,所以在网络这一层上可以认为大体上是可靠的。如果能进一步采取措施,比方说让每个节点都带上两个网口、连上两个局域网,那当然也有助于提高可靠性,但是在网络这一层上再提高可靠性的空间已经不大。

往上走一层,我们考虑节点主机硬件的可靠性。机器设备的可靠性是以平均无故障时间

(Mean Time Between Failure, MTBF)衡量的。不同档次的机器,其质量、价位都不同,MTBF 当然也就不同。Hadoop 集群的设计目标是基于采用普通的民用商品,目前国产服务器的 MTBF 可以达到 12 万小时,而磁盘阵列的 MTBF 一般是 5 万小时。为简化讨论,我们就以 12 万小时的 MTBF 估算。这样,假设集群的规模是 5000 个节点,那么我们可以预期平均每 24 小时就有一台机器发生故障,也就是平均每天都有一次故障。即使把集群的规模降到 500 个节点,那也还是平均每 10 天就有一台机器发生故障。显然,如果不采取任何措施,每一台机器的故障都有可能暴露于用户面前,这样的故障率是不可接受的,这决定了 Hadoop 的设计必须把容错乃至 HA 作为一个重点加以考虑。

这里还要说明一下,当我们考虑机器或集群的 MTBF 时,前提是运行的环境,包括电源、温度等等,是得到了绝对保证的。以电源为例,机器本身有个电源部分,这部分的可可靠性已经计算在机器的 MTBF 之内,但是机器外部的供电乃至整个机房的供电,是否得到保证呢?如果得不到保证,那么单纯谈论机器的 MTBF 就失去意义了。所以实际上也要把外部电源的 MTBF 一起计算进去,那样当然会使综合的 MTBF 下降,而且 MTBF 的计算也变得更复杂了。为简化讨论,我们在这里就不考虑这些因素,而假定其为绝对可靠了。

再考虑节点的宿主操作系统和 Java 虚拟机的可靠性。Hadoop 主要是用在 Linux 上,操作系统连同运行在此上的 Java 虚拟机都很稳定可靠,事实上这个环节的可靠性也几乎没有什么可以提高的空间,即便还有那么点不稳定也只能作为一种现实加以接受。

这样,网络、节点主机、运行环境、宿主操作系统和 Java 虚拟机,这些成分结合在一起构成了 Hadoop 的基础平台,这个基础平台的综合 MTBF 当然低于其中单台节点主机的 MTBF,不过我们并不关心精确的 MTBF 究竟是多少,因为上面只按节点主机估算的数字就已经是不可接受的了。

正因为这样,Hadoop 的设计中已经包含了一些容错措施。我们分 HDFS 和 YARN 两个层次加以考察。

在 HDFS 这一层上,文件的内容和文件目录是分开存储的。文件内容都存储在 DataNode 上,DataNode 是大量的;文件目录存储在 NameNode 上,NameNode 一般只有一个。文件内容之所以都分布存储在 DataNode 上,是 MapReduce 这个计算模式以及“数据在哪,计算就在哪”这个原则所决定的,要不然就回到了文件服务器加储值域网/局域网的架构。而文件目录之所以集中存储在 NameNode 上,其实是目录的本质所决定的,目录本来就是一种需要有集中管理的树形结构。设想如果这棵树变得很大的时候,NameNode 也许就不是只有一个了,但是这种集中式的树形结构并不会变。即使大到像互联网那样,也还需要有分级的 DNS,那实际上就是目录。所以,分布的数据即文件内容,和集中的目录,这是计算模型所决定的,问题是在这样的前提下如何提高系统的可靠性。

我们知道,HDFS 在数据的存储上引入了冗余,一个数据块被同时存储在几个(一般是三个)不同的节点上,形成同一数据块的多个复份(Replica)。这样,为简化讨论,姑且假定一个复份损坏的概率是千分之一,那么两个复份同时损坏的概率就只有百万分之一,三个复份同时损坏就几乎不可能了。NameNode 监视着所有数据块的复份,一旦发现某个数据块有一个复份缺失,就马上找一个完好的复份将其复制到另一个节点上,补上一个新的复份,以恢复预定的复份数量。在此期间用户对这个数据块的访问被引导到仍旧完好的那两个复份上,所以对故障不会有什么感觉。如果补上以后过一会儿缺失的那个复份又回来了(多半是因为其所在

的节点重启或修复了),那就从中删去一个,总之还是维持预定的复份数量。当然,当一个节点损坏时,因此而缺失的数据块复份可能是大量的,这时候会有大量的复制,但是这些复制操作是并发的,而且分布在许多不同的节点上,因而属于并行,由此而造成的流量也分布在网络的许多不同部位,NameNode 只是发布一下命令而已。这样的方案,就文件内容而言已经达到了容错和 HA 的要求,事实上恐怕也难有更可靠的方案了。

但是目录管理的情况有所不同,目录管理的操作是需要集中和排他的。如果目录的内容是只读的,不会改变(这意味着不会再创建或删除文件,也不会改变任何文件的内容),那倒可以有多个目录服务同时并存。但是,如果目录的内容可以改变,那就不能允许“七手八脚”而只能是集中和排他的了。进一步,对于 DataNode 和数据块复份的管理,当然也不能政出多门。所以对目录的管理是不允许有多个管理者同时并存的,要通过引入冗余来达到 HA 的目标,只能采用“热备份”,即“Hot Active/Standby”的方案。这种方案中,真正“在位”行使职权的即 Active 的管理者只有一个,但是有个作为备份即 Standby 的候补管理者时刻准备着,一旦发现 Active 的那个管理者发生故障,就立即顶上,把业务都接管过来,不用临时才开机和初始化,所以称为“热备”。当然,这种业务的接管和角色的倒换(Failover)需要是“无缝”的过程,要尽量让用户感觉不到。

不过,如果硬性规定非得有 NameNode 的热备份才能使用 Hadoop,那也不太合适,因为有些用户其实并没有那么高的要求。再说没有热备份的 NameNode 是不是就很脆弱了呢?那也未必。这是因为,就一个 500 节点的集群而言,其整体的 MTBF 固然是降到了 240 小时,以致平均每 10 天就有一台机器发生故障,但是就其中的每一台个别的机器而言,它的 MTBF 仍是 12 万小时,或者至少是数万小时(因机器质量而异),所以 NameNode 的 MTBF 也至少是数万小时,这对于有些用户很可能已经够了。正因为这样,文件系统的 HA 在 Hadoop 中是作为选项提供的,而具体的措施也就集中在 NameNode 的热备。

另一方面,即使没有热备,NameNode 在提高目录管理的容错能力方面也不是完全无所作为,Checkpoint 和 EditLog 的机制实际上也起不小的作用。

我们知道,Hadoop 的目录映像平时只是存在于内存中,其内容与磁盘上的映像并不总是同步的。这就像我们平时写磁盘文件,要 flush()一下才真的写入磁盘。但是如果突然断电,那么内存中的映像消失了,下次加电开机时只有磁盘中的映像还在,然而这个目录映像与这么多 DataNode 中的实际情况不一致,而且是不可恢复的了。这时候,即便有个 DataNode 报告,说我这儿有个复份属于几号数据块,但是 NameNode 的目录映像中却没有关于这个数据块的记录,因而就不知道它属于哪个文件,这样当然谈不上“容错”。

为克服这个问题,Hadoop 采取了 Checkpoint 加 EditLog 的措施,每当有改变文件系统内容的操作时,就在 EditLog 中记上一笔,这个 EditLog 就像是一本流水账,而且是记在磁盘上(也可以发送到另一个节点上)。这样,如果发生上述的情况,就可以拿磁盘上的映像作为基础,再按 EditLog 的内容重演一遍,就可以得到与实际情况相符的映像,这样就能“容错”了。读者也许会问,既然这流水账是在磁盘上,那为什么不把对于目录映像的修改直接写入磁盘?这是因为,比之往 EditLog 中添加一条记录,对目录映像的修改一般要复杂而漫长得多,首先在效率上就有影响。另一方面,EditLog 中的一条记录就是一道命令,它所对应的很可能不是单个的“原子”操作,而是一个需要维持原子性的操作过程,就像数据库操作中的 Transaction。对这样的操作,即便直接写入磁盘也未必能保证其一致性,所以其实数据库中 Transaction 的

实现也要使用这样的流水账。

在这种方法里,磁盘上的映像 FsImage 就是一个 Checkpoint,一个里程碑式的基准点、同步点,有了一个 Checkpoint 之后,NameNode 在相当长的时间内只是对内存中的目录映像操作,同时也对磁盘上的 EditLog 操作,直到关机。

下次开机的时候,NameNode 要从磁盘上装载目录映像 FSImage,那其实就是老的 Checkpoint,也许就是上次开机后所保存的映像,而自从上次开机后直到关机为止对于文件系统的所有改变都记录在 EditLog 文件中;将记录在 EditLog 中的操作重演于上一次的映像,就得到这一次的新的映像,将其写回磁盘就是新的 Checkpoint。

这个办法虽然简单却有缺点。可想而知,如果一次开机后要运行很长时间,则 EditLog 可能会很长,下次开机后生成原始映像的过程也会很长,可能会使人觉得开机初始化所需的时间长得难以忍受。对此的改进,自然是不必等下次开机,而是每当 EditLog 长到一定程度,或者每隔一定的时间,就做一次 Checkpoint。但是这样又带来一个问题,就是 NameNode 的负载可能太重了,会影响系统的性能。于是就有了 SecondaryNameNode 的需要,这相当于 NameNode 的助理,专替 NameNode 做 Checkpoint。当然,SecondaryNameNode 的负载相比之下是偏轻的。所以如果为 NameNode 配上了热备份,就可以让热备份兼职,而无须再有专职的 SecondaryNameNode。

可想而知,如果不给 NameNode 配上热备份,那么虽然也可以说是达到了容错的要求,却显然无法达到 HA(即高可用)的要求。试想如果 NameNode 出了故障,此时唯一的办法就是停下来把 NameNode 修好,此时整个 Hadoop 集群就不是 Available,而是不可用的了。

综上所述,Hadoop 在 HDFS 的设计中已经考虑了容错,但是其 NameNode 仍未达到 HA 的要求,解决问题的方法就是为其配上热备份,即 Standby。

比 Active/Standby 更高级的措施就是所谓“双活(Dual-Active)”甚至“多活”,包括“异地双活”。但是对于像 Hadoop 这样的地理上集中于一地,由同一管理者管理的系统而言一般并无这种必要。

再看 YARN 这一层。在 Hadoop 早期的设计中,这一层的主节点所扮演的角色被称为 JobTracker,意思是作业管理,实际上是把资源管理、作业的分解和指派调度及跟踪管理全都放在同一个节点上。当集群的规模较小、主节点的负载较轻的时候,这样也有好处,就是效率可以高一点。但是当集群的规模大到一定程度时,主节点的负担就太重了。另一方面,将所有作业的管理都放在主节点上,一旦主节点出了故障或因故暂时退出运行,这些作业就成了“群龙无首”,整个集群就难以维持运行了。所以,在后来的设计中把对于作业的管理都分散出去,每个作业都有个“应用管理者”AM,这就好比一个“课题组长”,每个 AM 所在的节点都是在创建作业,即“立项”的时候动态分配的。主节点则专责从事资源管理(以及受理用户的作业提交),所以称为“资源管理者”RM。这个设计方案就称为 YARN。这样,一方面是主节点的负载减轻了,另一方面系统的可用性也有所提高,因为就具体的作业而言一旦有了自己的 AM 并拿到了所需的资源以后,对于 RM 的依赖就不高了。至于说如果 AM 所在的节点损坏,或者某个承担计算任务的线程(例如某个 Mapper)崩溃或停滞,则最多也只是重启这个具体作业的执行,而并无全局的影响。更何况,无论是从前的 JobTracker 还是现在的 AM,都有所谓 Speculator 任务的安排,这就好比是个“预备队”,预先给它分配了必要的资源,让它随时准备增援前方,如果一切正常就不予动用,但是一旦发现某个具体的计算(线程)进度明显慢于预

期,就让这个 Speculator 顶上去将任务接管过来。显然,这也是通过冗余提高系统可靠性的思路,与 Active/Standby 是一样的道理。由此可见,YARN 这一层上也已经有了一些容错和提高系统可靠性的措施,剩下的薄弱环节就是 RM,这跟 HDFS 的情况是很相似的。

对于大数据处理(而不是存储),RM 显然是整个 Hadoop 集群是否可用的关键。不说别的,一旦 RM 节点失效,用户的作业提交就成了问题,在用户看来整个集群就不可用了。对于 Hadoop 集群,RM 是个“Single Point of Failure”,起着“一票否决”的作用。所以 Hadoop 对 RM 也提供 HA 选项,也采用 Active/Standby 的措施来达到 HA 的要求。

16.2 HDFS 的 HA 机制

如前所述,HDFS 有关 HA 的措施和机制集中在 NameNode 的热备,所以要了解 HDFS 的 HA 机制就必须了解 NameNode 的热备,反过来,明白了 NameNode 的热备怎样工作,也就明白了 HDFS 的 HA 机制是怎么回事。

Hadoop 的代码中就包含了 NameNode 的热备机制,在开机启动一个 NameNode 节点以后,用户可以通过 HA 管理命令行“hdfs haadmin”命令行指定将此节点用作常规的即 Active 的 NameNode,或是热备的即 Standby 的 NameNode。但是不管 Active 还是 Standby,所运行的都是 NameNode.class,JVM 起来后都会从 NameNode.main() 开始执行,然后都会调用 createNameNode(),看似意在创建 NameNode 对象。但是具体要做些什么,包括是否创建 NameNode,却要视名动命令行中的参数而定,如果参数中有“-bootstrapStandby”选项,那是要把本节点用作 NameNode 热备之前的准备工作,目的在于使本地的磁盘上具有与 ActiveNN,即 Active 的 NameNode 大致相同的文件系统映像。我们可以看一下 createNameNode() 的摘要:

```
[NameNode.main() > createNameNode()]
```

```
NameNode.createNameNode()
```

```
> ...
```

```
> switch (startOpt) {
```

```
> case FORMAT:    //只是文件系统的格式化,为担任 NameNode 做准备
```

```
>+ ...
```

```
>+ terminate()
```

```
> case FINALIZE:  //将 HDFS 文件系统现有的内容 finalize,做个认定
```

```
>+ ...
```

```
>+ terminate()
```

```
> case UPGRADEONLY: //HDFS 文件系统版本升级
```

```
>+ ...
```

```
> case ROLLBACK:  //HDFS 文件系统版本回滚,退回原来版本
```

```
>+ ...
```

```
>+ terminate()
```

```
> case RECOVER:   //故障(包括意外断电)后对 HDFS 文件系统的恢复
```

```
>+ ...
```

```

> case BOOTSTRAPSTANDBY: //如果是“-bootstrapStandby”选项
>+ String toolArgs[] = Arrays.copyOfRange(argv, 1, argv.length)
>+ int rc = BootstrapStandby.run(toolArgs, conf)
>+ terminate(rc) //terminate()结束当前进程,退出运行
>+ return null // avoid warning,这只是为避免 Java 编译发出警告
> default:
>+ return new NameNode(conf)
> }

```

以下我们用 ActiveNN 指活跃的、在位的 NameNode 对象,但请注意,并没有一个称为 ActiveNN 的类,那个类其实就是 NameNode;而 StandbyNN,则是指用作热备的 NameNode 对象,但是同样也没有 StandbyNN 这么一个类,那同样也是 NameNode。

注意,NameNode 对象的创建是在 default 分支下面,前面那些选项都不存在时才会跑到这里,而前面那些选项(其实还有更多)基本上都是像 BOOTSTRAPSTANDBY,做一点什么处理以后就通过 terminate() 结束当前进程,退出运行了。所以,如果命令行中是“-bootstrapStandby”选项,那就只是执行一下 BootstrapStandby.run(),然后就退出了:

```
[NameNode.main() > createNameNode() > BootstrapStandby.run()]
```

```

BootstrapStandby.run(String[] args) //BootstrapStandby 类的 run()函数
> parseArgs(args)
> parseConfAndFindOtherNN()
>> conf = getConfig()
>> nsId = DFSUtil.getNamenodeNameServiceId(conf)
>> if (!HAUtil.isHAEnabled(conf, nsId)) {
>>+ HadoopIllegalArgumentException("HA is not enabled for this namenode.")
//如果没有启用 HA 模式,而程序执行到了这里,就异常跳出了
>> }
>> ... //根据配置文件中的设置,找到有关其他 NameNode 的信息
> NameNode.checkAllowFormat(conf) //检查是否允许格式化
> InetAddress myAddr = NameNode.getAddress(conf)
> SecurityUtil.login(conf, DFS_NAMENODE_KEYTAB_FILE_KEY, ...) //要求用户登录
> return SecurityUtil.doAsLoginUserOrFatal(new PrivilegedAction<Integer>())
//以登录用户的权限执行以下程序
] run()
> try {
>+ return doRun()
> } catch (IOException e){
>+ throw new RuntimeException(e)
> }

```

后面实质性的操作都是要求用户登录之后以所登录的名义和权限,在一个实现了

PrivilegedAction 界面的对象的 run() 函数中完成的, 这里动态定义了这个类的 run() 函数, 代码中唯一的操作就是在一个 try{}catch() 框架中调用 doRun():

```
[NameNode.main() > createNameNode() > BootstrapStandby.run() => BootstrapStandby.doRun()]

BootstrapStandby.doRun()
> NamenodeProtocol proxy = createNNProtocolProxy()
    //与 Active 的 NameNode 建立联系, 创建代表着对方的 RPC proxy
> NamespaceInfo nsInfo = proxy.versionRequest() //通过 RPC 获取有关 Namespace 的信息
> if (!checkLayoutVersion(nsInfo)) {
>+ LOG.fatal("Layout version on remote node (" + nsInfo.getLayoutVersion() +
    ") does not match " + "this node's layout version (" +
    HdfsConstants.NAMENODE_LAYOUT_VERSION + ")")
>+ return ERR_CODE_INVALID_VERSION //版本不对
> }
> System.out.println("... About to bootstrap Standby ID " + nnId + " from:\n" + ...)
> imageTxId = proxy.getMostRecentCheckpointTxId() //RPC 调用
> curTxId = proxy.getTransactionID() //RPC 调用
> NNStorage storage = new NNStorage(conf, dirsToFormat, editUrisToFormat)
    //创建代表着这个热备 NameNode 上的存储设备(目录)的 NNStorage 对象
> // Check with the user before blowing away data.
> if (!Storage.confirmFormat(storage.dirIterable(null), force, interactive)) {
    //让用户确认是否真要格式化
>+ storage.close()
>+ return ERR_CODE_ALREADY_FORMATTED
> }
> // Format the storage (writes VERSION file)
> storage.format(nsInfo) //存储设备的格式化
> // Load the newly formatted image, using all of the directories (including shared edits)
> image = new FSImage(conf) //刚格式化的映像, 就是空白映像, 但是目录已经建好
> image.getStorage().setStorageInfo(storage)
> image.initEditLog(StartupOption.REGULAR)
> assert image.getEditLog().isOpenForRead() : "Expected edit log to be open for read"
> // Ensure that we have enough edits already in the shared directory
    to start up from the last checkpoint on the active.
> if (!skipSharedEditsCheck && !checkLogsAvailableForRead(image, imageTxId, curTxId))
    return ERR_CODE_LOGS_UNAVAILABLE
> image.getStorage().writeTransactionIdFileToStorage(curTxId)
> // Download that checkpoint into our storage directories. 从 Active NN 下载文件系统映像
> MD5Hash hash = TransferFsImage.downloadImageToStorage(
    otherHttpAddr, imageTxId, storage, true)
```

```
> image.saveDigestAndRenameCheckpointImage(NameNodeFile.IMAGE, imageTxId, hash)
//存储该 Checkpoint 的映像
```

完成了这些准备之后,用作 StandbyNN 的那个节点上,更确切地说是其宿主文件系统中,就有了与 ActiveNN 上相同的目录结构和文件系统映像,有了大致相同的起点。所谓相同的文件系统映像,是指 ActiveNN 上已经作为 Checkpoint 持久化存储下来的映像。但是 ActiveNN 还在运行中,它的 EditLog 可能还在变化。不过这也不要紧,StandbyNN 起来之后可以根据 EditLog 的内容在 Checkpoint 的基础上加以修正。所以,只能说 StandbyNN 所在的节点上有了与此刻的 ActiveNN 大致相同的起点。

此后,运行着 NameNode.class 的这个进程就退出运行了,因为 bootstrapStandby 已经完成,用户需要再次打入启动 NameNode 的命令行,这次不加“-bootstrapStandby”选项了。

由于不加“-bootstrapStandby”或别的某些选项,这次就真正创建 NameNode 对象了,所以会执行 NameNode 的构造方法,我们就从这里开始。

```
NameNode.NameNode() //NameNode 的构造方法
> String nsId = getNameServiceId(conf)
> String namenodeId = HAUtil.getNameNodeId(conf, nsId)
> this.haEnabled = HAUtil.isHAEnabled(conf, nsId)
> startOpt = getStartupOption(conf)
> HAAState state = createHAAState(startOpt) //确定本节点是 Active 还是 Standby
>> if (!haEnabled || startOpt == StartupOption.UPGRADE
    || startOpt == StartupOption.UPGRADEONLY) return ACTIVE_STATE
>> else return STANDBY_STATE
> this.allowStaleStandbyReads = HAUtil.shouldAllowStandbyReads(conf)
> this.haContext = createHAContext()
> initializeGenericKeys(conf, nsId, namenodeId)
> initialize(conf) //NameNode 节点的初始化,Active 和 Standby 一样
> state.prepareToEnterState(haContext)
> state.enterState(haContext) == HAAState.enterState(HAContext context) //分道扬镳
    == StandbyState.enterState(haContext)
>> context.startStandbyServices() //在 ActiveNN 上是 startActiveServices()
```

注意这里 createHAAState()所返回的 ACTIVE_STATE 或 STANDBY_STATE,这二者分别是 NameNode 内部的 ActiveState 和 StandbyState 类对象,而不是字符串常数。作为 NameNode 的结构成分,这两个对象是在创建 NameNode 对象时就创建好的,二者都实现了 HAAState 界面。这样,在调用 state.enterState()的时候,就会因 state 的不同而调用不同的 enterState()。

从代码中可见,如果系统的配置是 haEnabled,启动时又未使用 upgrade 或 upgradeonly 选项,那么 NameNode 就会进入 Standby 状态。假定集群中有两个节点都在启动 NameNode,就都会进入 Standby 状态。那么究竟谁会成为 Active 呢? 这有手动和自动两种方法。手动就是由系统管理员向其中之一打入 HA 管理命令使其变成 Active,自动则要有个 ZooKeeper 节点进行管理,本章后面会有介绍。


```
has not been upgraded yet. You should restart this NameNode with the '" +
StartupOption.BOOTSTRAPSTANDBY.getName() +
"' option to bring this NN in sync with the other.")
```

```
> ++ }
> + }
> + editLog.recoverUnclosedStreams()
> } else {
    // This NN is HA and we're not doing an upgrade. HA 已启用, 但这是 StandbyNN
> + editLog.initSharedJournalsForRead() // 只需要读 Journal
> +> initJournals(this.sharedEditsDirs)
> +> state = State.OPEN_FOR_READING
> }
```

注意调用 `initJournals()` 时的参数。对于 `ActiveNN`, 调用的参数是 `editsDirs`, 这是它自己的一个或几个日志文件 (Journal) 目录; 对于 `StandbyNN`, 调用的参数是 `sharedEditsDirs`, 这是要共享别人 (即 `ActiveNN`) 的日志文件目录。

```
[NameNode.initialize() > NameNode.loadNamesystem() > FSNamesystem.loadFromDisk()
> FSImage.loadFSImage() > FSImage.initEditLog() > ... > FSEditLog.initJournals()]
```

```
FSEditLog.initJournals(List<URI> dirs)
> minimumRedundantJournals = // 从配置文件获取设定值, 默认为 1
    conf.getInt(..., DFS_NAMENODE_EDITS_DIR_MINIMUM_DEFAULT)
> journalSet = new JournalSet(minimumRedundantJournals) // 创建一个集合
> for (URI u : dirs) { // 对于参数给定的每一个文件目录
> + boolean required = FSNamesystem.getRequiredNamespaceEditsDirs(conf).contains(u)
> + if (u.getScheme().equals(NNStorage.LOCAL_URI_SCHEME)) { // 存放在本地文件中
> ++ StorageDirectory sd = storage.getStorageDirectory(u)
> ++ if (sd != null) {
> +++ fjm = new FileJournalManager(conf, sd, storage) // 管理和代表本地的 Journal 文件
> +++ journalSet.add(fjm, required, sharedEditsDirs.contains(u))
> ++ }
> + } else {
> ++ JournalManager jm = createJournal(u) // 跨节点外挂, 实际上是 QuorumJournalManager
> ++> Class<? extends JournalManager> clazz = getJournalClass(conf, uri.getScheme())
> ++>> key = DFSConfigKeys.DFS_NAMENODE_EDITS_PLUGIN_PREFIX + "." + uriScheme
> ++>> clazz = conf.getClass(key, null, JournalManager.class)
> ++> Constructor<? extends JournalManager> cons =
    clazz.getConstructor(Configuration.class, URI.class, NamespaceInfo.class)
> ++> return cons.newInstance(conf, uri, storage.getNamespaceInfo())
> ++ journalSet.add(jm, required, sharedEditsDirs.contains(u))
```



```
>+ }
> }
```

如果不启用 HA,那么 Journal 文件所在的目录只供唯一的 NameNode 节点使用,这时候它的 scheme 应该是 LOCAL_URI_SCHEME。但是启用了 HA 就不同了,此时的 Journal 文件需要让 StandbyNN 共享,这时候的 scheme 就不是 LOCAL_URI_SCHEME 了。这里的 DFS_NAMENODE_EDITS_PLUGIN_PREFIX 是字符串“dfs.namenode.edits.journal-plugin”,这被用作前缀,后面再跟上具体 scheme 的名称,就是配置文件中使用的属性名。

配置文件 hdfs-default.xml 中有个属性“dfs.namenode.edits.journal-plugin.qjournal”,就是用来配置采用何种 JournalManager,其实是实现了这个 interface 的类。实际设置的值通常是“org.apache.hadoop.hdfs.qjournal.client.QuorumJournalManager”,所以这里所创建的 JournalManager 实际上是 QuorumJournalManager,这跟 FileJournalManager 一样,是用来记录文件系统交易(即引起文件系统变化的操作)的,只不过前者做成一个服务器,而后者只是把流水账记在本地的(宿主)文件系统中。

完成了作为 NameNode 的初始化,下面就是对 HASTate.enterState()的调用,StandbyNN 与 ActiveNN 在这里就分道扬镳了。对于 Standby 节点,所调用的是 StandbyState.enterState():

```
StandbyState.enterState(HAContext context)
>context.startStandbyServices() == NameNodeHAContext.startStandbyServices()
```

这里的 HAContext 是个 interface,实现了这个 interface 的类是定义于 NameNode 类内部的 NameNodeHAContext。

```
class NameNode.NameNodeHAContext implements HAContext {}
] startActiveServices()
  > namesystem.startActiveServices()
  > startTrashEmptier(conf)
] startStandbyServices()
  > namesystem.startStandbyServices(conf)
```

所以,对于 StandbyNN,在这一步上调用的是 FSNamesystem.startStandbyServices():

```
[NameNode()>StandbyState.enterState() => FSNamesystem.startStandbyServices()]
```

```
FSNamesystem.startStandbyServices()
> if (!getFSImage().editLog.isOpenForRead()) {
    //During startup, we're already open for read.
>+ getFSImage().editLog.initSharedJournalsForRead()
> }
> blockManager.setPostponeBlocksFromFuture(true)
>> blockManager.shouldPostponeBlocksFromFuture = true
    //StandbyNN 可能收到 DataNode 的块报告早于从 ActiveNN 获得信息
    //这个变量表示对于来自 DataNode 的报告要推迟处理。见 processReportedBlock()
> dir.disableQuotaChecks() //Disable quota checks while in standby. 这是 ActiveNN 的事
```

```

> editLogTailer = new EditLogTailer(this, conf)
    //创建 EditLogTailer 对象及其线程,专门盯住 ActiveNN 的 EditLog
> editLogTailer.start()    //启动 EditLogTailer 线程
> if (standbyShouldCheckpoint) {
>+ standbyCheckpointner = new StandbyCheckpointner(conf, this)
    //创建 StandbyCheckpointner 对象及其线程,负责定期做 Checkpoint
>+ standbyCheckpointner.start()
> }

```

StandbyNN 在平时即不需要接管 ActiveNN 的时候,主要有两方面的活动:第一,是通过其 EditLogTailer 线程紧盯 ActiveNN 的 EditLog,在其本地的文件系统映像中重演 ActiveNN 上对文件系统的种种变动,使其文件系统映像尽量与 ActiveNN 上保持一致;第二,如果没有配置说不要,就由其 StandbyCheckpointner 线程定期做 Checkpoint。至于 initSharedJournalsForRead(),那只是为这两个线程创造工作条件。

我们先看看 EditLogTailer 的摘要:

```

class EditLogTailer {}
] EditLogTailerThread tailerThread //这个线程是 NameNode 热备节点的主体
] FSNamesystem namesystem
] FSEditLog editLog
] InetAddress activeAddr
] NamenodeProtocol cachedActiveProxy
] EditLogTailer(FSNamesystem namesystem, Configuration conf) //构造方法
> this.tailerThread = new EditLogTailerThread() //创建 EditLogTailerThread 线程
> this.conf = conf
> this.namesystem = namesystem //这是 StandbyNN 上的 FSNamesystem
> this.editLog = namesystem.getEditLog()
> logRollPeriodMs = conf.getInt(DFSConfigKeys.DFS_HA_LOGROLL_PERIOD_KEY,
    DFSConfigKeys.DFS_HA_LOGROLL_PERIOD_DEFAULT) * 1000
> this.activeAddr = getActiveNodeAddress()
> sleepTimeMs = conf.getInt(DFSConfigKeys.DFS_HA_TAILEDITS_PERIOD_KEY,
    DFSConfigKeys.DFS_HA_TAILEDITS_PERIOD_DEFAULT) * 1000
] class EditLogTailerThread extends Thread {}
]] run()
> act = new PrivilegedAction<Object>() //动态定义一个实现 PrivilegedAction 界面的类
] run() //它的 run()函数
> doWork()
> SecurityUtil.doAsLoginUserOrFatal(act) //以登录用户的身份运行这个线程

```

显然,创建 EditLogTailer 对象实际上就是创建 EditLogTailerThread 线程,这个线程是以登录用户的身份运行,其 run()函数唯一的操作就是 doWork():

```

[EditLogTailerThread.run() > EditLogTailer.doWork()]

```

```

EditLogTailer.doWork()
> while (shouldRun) {
>+ if (tooLongSinceLastLoad() && lastRollTriggerTxId < lastLoadedTxnId) {
>++ triggerActiveLogRoll() //如果有很长时间没有滚进 EditLog,就触发一下
>++> getActiveNodeProxy().rollEditLog() //对 ActiveNN 的 RPC 调用
>++> lastRollTriggerTxId = lastLoadedTxnId
>+ }
>+ doTailEdits() //按 EditLog 的记载更新本地的文件系统映像
>+> FSImage image = namesystem.getFSImage() //本地的文件系统映像
>+> lastTxnId = image.getLastAppliedTxId()
>+> Collection<EditLogInputStream> streams =
        editLog.selectInputStreams(lastTxnId + 1, 0, null, false)
        //得到读取 ActiveNN 的 EditLog 的输入流
>+>> streams = new ArrayList<EditLogInputStream>()
>+>> Preconditions.checkState(journalSet.isOpen(),
        "Cannot call selectInputStreams() on closed FSEditLog")
        //如果不能访问 ActiveNN 的 EditLog 就报警
>+>> selectInputStreams(streams, fromTxId, inProgressOk)
>+>> checkForGaps(streams, fromTxId, toAtLeastTxId, inProgressOk)
        //通过比较两地的 Transaction ID,可知与 ActiveNN 上的文件系统映像差距有多大
>+> editsLoaded = image.loadEdits(streams, namesystem)
        == FSImage.loadEdits(Iterable<EditLogInputStream> editStreams, FSNamesystem target)
        //在本地文件系统映像上加载(重演)来自 ActiveNN 的 EditLog
>+> if (editsLoaded > 0) lastLoadTimestamp = now() //记下这次加载的时间
>+> lastLoadedTxnId = image.getLastAppliedTxId()
        //EditLog 中记载的最后一次交易(Transaction)的 ID
>+ Thread.sleep(sleepTimeMs) //睡眠一段时间
> } //end while

```

简而言之,就是 StandbyNN 上的这个线程过一会儿就去读 ActiveNN 的 EditLog,将这个 EditLog 中记录的操作在 StandbyNN 本地的文件系统映像上加以重演。注意 StandbyNN 本身是没有 EditLog 的,因为 EditLog 中记录的操作都来自 Client,而 Client 根本就不知道 StandbyNN 的存在。

经过这样一次 FSImage.loadEdits(),StandbyNN 这边的文件系统映像就与 ActiveNN 上基本一致了。稍差几个交易(Transaction)也不要紧,只要这些记录在 EditLog 中就漏不了,因为所有的 Transaction 都是单调递增编号的,只要记住这一次处理到几号 Transaction,下一次就从下一个开始。不过这只是使两边内存中的文件系统映像基本一致了,这个线程并没有把文件系统映像写到磁盘中,如果写入磁盘就成了一个 Checkpoint。

前面讲过,处于热备状态的 StandbyNN 主要做两件事:一件是对于 EditLog 的跟进处理,这是必需的;另一件是取代 SecondaryNN 的 Checkpoint 处理,这是可选的,取决于配置文件中对于属性“dfs.ha.standby.checkpoints”的设置,但是默认值为“true”,所以实际上也属于标配,再说一般而言也没有什么理由不要 Checkpoint 处理。也正因为这样,有了 StandbyNN 就不需要有 SecondaryNN 了。

如果存储在内存中的信息永远不会消失,那就根本不需要 Checkpoint。但是现实是内存中的信息一断电就没有了,所以得要写到磁盘上去。另一方面,如果写磁盘的速度很快,而且像内存一样可以按字节寻址写入,那么反过来内存中就不需要有文件系统映像了,直接写入磁盘就行,可不幸的是磁盘的读写相对而言很慢,而且只能是按扇区存储。这样,把每一次 Transaction,即对于文件系统目录的改动都实时记入磁盘是不现实的。但是如果一直不写磁盘,直到关机开机或重启后再在上次的映像中重演 EditLog 中积累的全部交易记录,虽然逻辑上并无问题,但是这个 EditLog 加载(即重演)的过程就太长了。所以,比较好的办法是折中一下,过一会儿,适当长的一会儿,就写一次磁盘。这样,写一次磁盘,就得到一个 Checkpoint,这是一个里程碑式的基准点。当然,从原理上说,NameNode 自己完全可以做这个事,但是实际上却常常忙不过来。所以在 Hadoop 尚不支持 HA 的时候就为 NameNode 设计了一个 SecondaryNameNode 节点,即 NameNode 助理,让它专门替 NameNode 做 Checkpoint,其功能跟我们现在看到的很相似,只是 SecondaryNameNode 没有在发生故障时接管 NameNode 功能的意向和能力,而 StandbyNN 则时刻准备着接管 ActiveNN 的服务和功能。

所以,在为 NameNode 选用 HA 选项,为其配上热备份之后,这部分功能就并入了 NameNode 的 Standby 节点上,而不再需要有 NameNode 助理即 SecondaryNN。

明白了这些背景,我们看 StandbyNN 上 StandbyCheckpoint 类的摘要:

```
class StandbyCheckpointer {}

] FSNamesystem namesystem
] CheckpointerThread thread
] ThreadFactory uploadThreadFactory
] URL activeNNAddress
] URL myNNAddress
] StandbyCheckpointer(Configuration conf, FSNamesystem ns) //构造方法
    > this.namesystem = ns
    > this.conf = conf
    > this.checkpointConf = new CheckpointConf(conf)
    > this.thread = new CheckpointerThread() //创建 CheckpointerThread 线程
    > this.uploadThreadFactory = new ThreadFactoryBuilder()
        .setDaemon(true).setNameFormat("TransferFsImageUpload- %d").build()
        //在 doCheckpoint() 内部用来创建上传 checkpoint 映像文件的线程
    > setNameNodeAddresses(conf)
    >> myNNAddress = getHttpAddress(conf)
    >> Configuration confForActive = HAUtil.getConfForOtherNode(conf) //获取对方的 conf
    >>> String nsId = DFSUtil.getNamenodeNameServiceId(myConf)
```

```

>>>> String otherNn = getNameNodeIdOfOtherNode(myConf, nsId) //ActiveNN 的节点名
>>>> // Look up the address of the active NN.
>>>> confForOtherNode = new Configuration(myConf)
>>>> NameNode.initializeGenericKeys(confForOtherNode, nsId, otherNn)
>>>> return confForOtherNode
>> activeNNAddress = getHttpAddress(confForActive) //获取 ActiveNN 的网址
] class CheckpointerThread extends Thread {}
]] run()
    > act = new PrivilegedAction<Object>()
        ] run()
            > doWork()
        > SecurityUtil.doAsLoginUserOrFatal(act)

```

先大致看一下这个类的构造方法 StandbyCheckpointer()。这里最重要的操作当然是 CheckpointerThread 的创建,因为实际的 Checkpoint 操作都是由这个线程实施的。另外还创建了一个 ThreadFactoryBuilder 对象 uploadThreadFactory,以便有需要时就立即生出一个线程从事 Checkpoint 映像文件的上传。鉴于本节点是个 Standby 的 NameNode,我们当然需要知道相应 ActiveNN 的节点名和地址,这里的 setNameNodeAddresses()就是为这个目的。作为 NameNode,配置文件 hdfs-default.xml 中有几个属性与此直接相关:“dfs.nameservice.id”下面是 NameNode 所属的 nameservice,也就是 namespace;“dfs.ha.namenode.id”就是本 NameNode 的节点名;而以“dfs.ha.namenodes”为前缀(后面是 nameservice 的名称)的属性名下面则是以逗号分隔的一组节点名,对于 Active/Standby 而言应该是两个节点名,其中之一是本节点,另一个就是 Active 的那个 NameNode。

有了这些以后,下面就是 CheckpointerThread 这个线程的事了,只是在需要上传 Checkpoint 映像文件时需要 uploadThread 的协助。从上面的摘要可知,CheckpointerThread 这个线程的主体也叫 doWork(),注意不要与前面的那个 EditLogTailer.doWork()混淆。

```
[StandbyCheckpointer.CheckpointerThread.run() > StandbyCheckpointer.doWork()]
```

```

StandbyCheckpointer.doWork()
> checkPeriod = 1000 * checkpointConf.getCheckPeriod()
> lastCheckpointTime = monotonicNow()
> while (shouldRun) {
>+ needRollbackCheckpoint = namesystem.isNeedRollbackFsImage()
                                //检查是否需要做一个 RollbackCheckpoint
>+ if (!needRollbackCheckpoint) {
>++ Thread.sleep(checkPeriod) //睡眠一段时间,默认为 60 秒
>++ if (!shouldRun) break
>+ }
>+ uncheckpointed = countUncheckpointedTxns() //数一下 EditLog 中已经有了多少个交易
>+ secsSinceLast = (now - lastCheckpointTime)/1000 //以及已经有了多久

```

```

>+ needCheckpoint = needRollbackCheckpoint //如果需要 Rollback 就不管数量和时间了
>+ if (needCheckpoint) {
>+ LOG.info("Triggering a rollback fsimage for rolling upgrade.")
>+ } else if (uncheckpointed >= checkpointConf.getTxnCount()) { //如果交易数量已多
>+ LOG.info("Triggering checkpoint because there have been uncheckpointed " + ...)
>+ needCheckpoint = true
>+ } else if (secsSinceLast >= checkpointConf.getPeriod()) { //如果时间已久
>+ LOG.info("Triggering checkpoint because it has been " + ... + " seconds " + ...)
>+ needCheckpoint = true
>+ }
>+ if (now < preventCheckpointsUntil) {
>+ LOG.info("But skipping this checkpoint since we are about to failover!")
>+ canceledCount ++
>+ continue //如果尚在禁止期内,则跳过本次操作
>+ }
>+ assert canceler == null
>+ canceler = new Canceler() //在 doCheckpoint()里面会用到
>+ if (needCheckpoint) { //如果需要做 Checkpoint
>+ doCheckpoint()
>+ if (needRollbackCheckpoint && namesystem.getFSImage().hasRollbackFSImage()) {
>+ //如果这个 Checkpoint 是因为需要 Rollback 而做
>+ namesystem.setCreatedRollbackImages(true) //表示已经创建了为 Rollback 准备的映像
>+ namesystem.setNeedRollbackFsImage(false) //表示不再需要创建 Rollback 所需的映像
>+ }
>+ lastCheckpointTime = now
>+ }
> } //end while

```

显然,这里的 while 循环就是这个线程的主循环。在每一轮的循环中,这个线程先睡眠一下,然后看看自上次 Checkpoint 以来是否已经有了很多交易(Transaction,指改变 HDFS 文件系统内容的操作,默认为不超过 1000000 次),或者已经过去了很长时间(默认为 3600 秒)。只要满足这两个条件之一,就应该做一个新的 Checkpoint 了。此外,如果需要做一个 RollbackCheckpoint,就是为可能会发生的 Rollback 准备好一个 Checkpoint(这种特殊情况发生在需要版本升级的时候),那就特事特办、马上做,连睡眠也跳过,更不用管那两个条件了。如前所述,所谓 Checkpoint 就是文件系统在某个时间点上的映像,如果 StandbyNN 在 EditLog 中看到一个 OP_ROLLING_UPGRADE_START 操作,就说明 ActiveNN 上正在进行一次版本的滚进升级,可是版本升级是有可能失败的,理应在开始升级之前把原有的映像保存下来成为一个 Checkpoint,以备万一需要恢复。但是如前所说,让 ActiveNN 在此之前保存文件系统映像往往是忙不过来的。现在有了 StandbyNN,这事情就由它承担了。显然,当 StandbyNN 的 EditLogTailer 线程从 EditLog 中逐条记录读出并重演,到看到这条

OP_ROLLING_UPGRADE_START操作记录时,其文件系统映像恰好就是 ActiveNN 上开始升级前夕的映像,把这个映像保存下来,就是为可能需要的回滚准备下一个 Checkpoint。此时 EditLogTailer 线程会通过 setNeedRollbackFsImage(true)设置一个变量,让专做 Checkpoint 的 StandbyCheckpointer 线程知道。

这样,当确定了要做一个 Checkpoint 时,就调用 doCheckpoint():

```
[StandbyCheckpointer.CheckpointerThread.run() > StandbyCheckpointer.doWork()
> doCheckpoint()]

StandbyCheckpointer.doCheckpoint()
> assert canceler != null
> assert namesystem.getEditLog().isOpenForRead()
> FSImage img = namesystem.getFSImage()
> prevCheckpointTxId = img.getStorage().getMostRecentCheckpointTxId()
    //这是上一个 Checkpoint 中最后的那个 TxId
> thisCheckpointTxId = img.getLastAppliedOrWrittenTxId() //这是 EditLog 中最新的 TxId
> if (thisCheckpointTxId == prevCheckpointTxId) return //没有新的操作,无须保存
> if (namesystem.isRollingUpgrade()
    &&!namesystem.getFSImage().hasRollbackFSImage()) {
>+ imageType = NameNodeFile.IMAGE_ROLLBACK //这是专供版本回滚的
> } else {
>+ imageType = NameNodeFile.IMAGE //这是供故障恢复的
> }
> img.saveNamespace(namesystem, imageType, canceler)
    //将文件系统映像保存在磁盘文件中
> txid = img.getStorage().getMostRecentCheckpointTxId()
> assert txid == thisCheckpointTxId //验证新保存的 Checkpoint 是否正确
> // Save the legacy OIV image, if the output dir is defined.
> String outputDir = checkpointConf.getLegacyOivImageDir()
    //OIV 是“Offline Image Viewer”的缩写,是老版 Hadoop 采用的格式,现已不用
> if (outputDir != null &&!outputDir.isEmpty()) {
>+ img.saveLegacyOIVImage(namesystem, outputDir, canceler)
> }
> ExecutorService executor = Executors.newSingleThreadExecutor(uploadThreadFactory)
    //创建一个线程,用来执行下面创建的这个 Callable
> c = new Callable<Void>() //创建一个动态定义的 Callable
    ] call()
    > TransferFsImage.uploadImageFromStorage(activeNNAddress, conf,
        namesystem.getFSImage().getStorage(), imageType, txid, canceler)
> Future<Void> upload = executor.submit(c) //让刚创建的线程运行这个 Callable
> executor.shutdown()
```

```
> upload.get()
```

摘要中加了足够的注释,就不用另外再解释了。最后创建的这个线程,目的是要把 StandbyNN 刚保存的这个 checkpoint,即文件系统映像文件上传到 ActiveNN 去:

```
[StandbyCheckpointeer.CheckpointerThread.run() > StandbyCheckpointeer.doWork()
> doCheckpoint() > TransferFsImage.uploadImageFromStorage()]
```

```
TransferFsImage.uploadImageFromStorage(URL fsName, Configuration conf,
    NNStorage storage, NameNodeFile nnf, long txid, Canceler canceler)
> // Uploads the imagefile using HTTP PUT method
> url = new URL(fsName, ImageServlet.PATH_SPEC)
> uploadImage(url, conf, storage, nnf, txid, canceler) //采用 HTTP PUT 上传映像文件
    //显然,ActiveNN 上有个 HTTP server
>> File imageFile = storage.findImageFile(nnf, txId)
>> uriBuilder = new URIBuilder(url.toURI())
>> Map<String, String> params =
    ImageServlet.getParamsForPutImage(storage, txId, imageFile.length(), nnf)
>> for (Entry<String, String> entry : params.entrySet()) {
>>+ uriBuilder.addParameter(entry.getKey(), entry.getValue())
>> }
>> URL urlWithParams = uriBuilder.build().toURL()
>> connection = (HttpURLConnection) connectionFactory.openConnection(urlWithParams,
    UserGroupInformation.isSecurityEnabled())
>> connection.setRequestMethod("PUT")
>> connection.setDoOutput(true)
>> chunkSize = conf.getInt(DFSConfigKeys.DFS_IMAGE_TRANSFER_CHUNKSIZE_KEY,
    DFSConfigKeys.DFS_IMAGE_TRANSFER_CHUNKSIZE_DEFAULT)
>> if (imageFile.length() > chunkSize) { // chunkSize 的默认值是 64KB
    // 如果文件长度大于此值,就应采用流模式,以免减小中间缓冲
    // 采用流模式后文件长度甚至可达 2GB 以上
>>+ connection.setChunkedStreamingMode(chunkSize)
>> }
>> setTimeout(connection)
>> ImageServlet.setVerificationHeadersForPut(connection, imageFile)
>> writeFileToPutRequest(conf, connection, imageFile, canceler)
    //Write the file to output stream.
>> responseCode = connection.getResponseCode() //读回对方响应
>> if (responseCode != HttpURLConnection.HTTP_OK) {
>>+ HttpPutFailedException(connection.getResponseMessage(), responseCode)
>> }
```

```
>> if (connection != null) connection.disconnect()
```

上面讲的是 StandbyNN 运行时的活动,但这只是与 ActiveNN 不同的那一部分,事实上另有大量的活动是相同的。我们看到,不管是 Active 还是 Standby,两种 NameNode 都调用了同样的初始化过程,有着基本相同的结构成分。

那么,ActiveNN 上的活动,特别是 ActiveNN 上有而 StandbyNN 上没有的活动,又是怎样的呢? 我们不妨看一下 startActiveServices():

```
[NameNode.NameNode() > FSNamesystem.startActiveServices()]

FSNamesystem.startActiveServices()
> editLog = getFSImage().getEditLog() //从宿主文件系统或服务器找到 EditLog
> if (!editLog.isOpenForWrite()) {
>+ editLog.initJournalsForWrite() //建立写 EditLog 的通道
//During startup, we're already open for write during initialization.
>+ editLog.recoverUnclosedStreams() // May need to recover
>+ LOG.info(
    "Catching up to latest edits from old active before taking over writer role in edits logs")
>+ editLogTailer.catchupDuringFailover()
>+ SecurityUtil.doAsLoginUser (new PrivilegedExceptionAction<Void>())
    ] run()
    > doTailEdits() //按 EditLog 的记载更新文件系统映像
>+ blockManager.setPostponeBlocksFromFuture(false) //在 StandbyNN 中是 true
>+ blockManager.getDatanodeManager().markAllDatanodesStale()
>+ blockManager.clearQueues()
>+ blockManager.processAllPendingDNMessages()
>+ if (!isInSafeMode()) {
>++ initializeReplQueues()
>+ }
>+ nextTxId = getFSImage().getLastAppliedTxId() + 1
>+ editLog.setNextTxId(nextTxId)
>+ getFSImage().editLog.openForWrite() //打开 EditLog 文件供写入
> }
> dir.enableQuotaChecks()
> if (haEnabled) {
>+ leaseManager.renewAllLeases()
//while we were in standby mode, the leases weren't getting renewed on this NN.
> }
> leaseManager.startMonitor() //Lease 是对 Client 而言的,所以 StandbyNN 不用操心这个
> startSecretManagerIfNecessary()
> this.nnrmtread = new Daemon(new NameNodeResourceMonitor())
```

```

//ResourceMonitor required only at ActiveNN.
> nrmthread.start()
> nnEditLogRoller = new Daemon(new NameNodeEditLogRoller(
    editLogRollerThreshold, editLogRollerInterval))
    //创建 NameNodeEditLogRoller 线程
    //默认的动作门槛值是 200 万条交易记录,或间隔时间达到 5 分钟
> nnEditLogRoller.start() //启动这个线程
> if (lazyPersistFileScrubIntervalSec > 0) {
>+ lazyPersist = new LazyPersistFileScrubber(lazyPersistFileScrubIntervalSec) //Runnable
>+ lazyPersistFileScrubber = new Daemon(lazyPersist) //创建线程
    // Daemon to periodically scan the namespace for lazyPersist files
    // with missing blocks and unlink them.
>+ lazyPersistFileScrubber.start()
> }
> cacheManager.startMonitorThread()
    //创建专管各 DataNode 上数据块缓存的 CacheReplicationMonitor 线程
> blockManager.getDatanager().setShouldSendCachingCommands(true)

```

注意这里的 `catchupDuringFailover()`, 这个函数的操作实际上是 `doTailEdits()`, 那就是我们在前面看到 StandbyNN 上的 `EditLogTailer` 线程每过一会儿就要做一次的事情。ActiveNN 初始化的时候, 内存中的 `FSImage` 是从磁盘上的最后一个 checkpoint 文件读入的, 但是这个 checkpoint 未必包含了所有发生过的文件系统映像变动, 所以还要再看看 `EditLog`, 如果有这样的操作就在最后一个 checkpoint 的基础上重演一下。这个函数为什么叫 `catchupDuringFailover()` 呢? `Failover` 是指当原来的 ActiveNN 发生故障, 而 StandbyNN 加以接管时的倒换。此时 StandbyNN 的文件系统映像与 ActiveNN 上的可能存在着差距, 所以需要 `catchup` 即“追赶上”ActiveNN 的文件系统映像, 方法就是重演那些尚未重演过的 `EditLog` 记录。当然, 如果实际上不存在差距, 那么所谓“追赶”也就没有什么实质的操作。

最后创建的这几个线程, 像 `NameNodeResourceMonitor` 的作用是不言自明的, 但是对 `NameNodeEditLogRoller` 这个线程需要作点说明:

```

NameNodeEditLogRoller.run()
> while (fsRunning && shouldRun) {
>+ editLog = getFSImage().getEditLog()
>+ numEdits = editLog.getLastWrittenTxId() - editLog.getCurSegmentTxId()
>+ if (numEdits > rollThreshold) {
>++ rollEditLog() == FSNamesystem.rollEditLog() // FSNamesystem 层面的滚进
>++> checkSuperuserPrivilege() //得有 Superuser 的权限
>++> checkOperation(OperationCategory.JOURNAL) //得有 JOURNAL 操作的权限
>++> checkNameNodeSafeMode("Log not rolled") //NameNode 处于 SafeMode 时不允许
>++> getFSImage().rollEditLog() == FSImage.rollEditLog() // FSImage 层面的滚进
>++>> getEditLog().rollEditLog() //Finalizes the current edit log and opens a new log segment.

```

```

        == FSEditLog.rollEditLog() // FSEditLog 的滚进
>+>+>+> endCurrentLogSegment(true) //结束当前日志段,关闭该文件
        == FSEditLog.endCurrentLogSegment(boolean writeEndTxn)
>+>+>+> nextTxId = getLastWrittenTxId() + 1
>+>+>+> startLogSegment(nextTxId, true) //开始下一个日志段,创建文件
        == FSEditLog.startLogSegment(long segmentTxId, boolean writeHeaderTxn)
>+>+>+> assert curSegmentTxId == nextTxId
>+>+> // Record this log segment ID in all of the storage directories
>+>+> storage.writeTransactionIdFileToStorage(getEditLog().getCurSegmentTxId())
>+>+>+> for (StorageDirectory sd : storageDirs) { //在每个存储目录中
>+>+>+>+ writeTransactionIdFile(sd, txid)
            //写一个名为“seen_txid_nnnnnnnnnnn”的文件,txid 号码长达 19 位数字
>+>+>+>+> File txIdFile = getStorageFile(sd, NameNodeFile.SEEN_TXID)
>+>+>+>+> PersistentLongFile.writeFile(txIdFile, txid) //把下个日志段中的首个 txid 写入文件
>+>+>+> }
>+>+> return new CheckpointSignature(this) //用于 Checkpoint 文件
>+ }
>+ Thread.sleep(sleepIntervalMs)
> } //end while

```

这个线程过一会儿就检查一下 EditLog,看是否积累了太多的操作记录,如果是的话就进行一次 FSNamesystem 层面的 rollEditLog(),就是 EditLog 的滚动前进。从原理上讲,EditLog 中的记录是只进不出的,一旦写入 EditLog 就永久存在,所以 EditLog 的大小是单调增长的。但是如果把所有的历史记录都保存在同一个文件中,则这个文件可能就太大了,所以积累到一定程度就得另写一个 EditLog 文件,或者甚至写到另一个日志服务器中。这样的一个 EditLog 文件或其他形式的存储单位,逻辑上就成为 EditLog 的一个“段(Segment)”。而所谓 EditLog 的滚进(roll),就是另起一段,一般是另起一个 EditLog 文件。此外,还要在每个存储目录中留下一个名为“seen_txid_nnnnnnnnnnn”的文件,表示已经有了一个以此 txid 开头的 EditLog 段。

16.3 NameNode 的倒换

StandbyNN 的作用不仅仅是替 ActiveNN 做 Checkpoint,更重要的是它还时刻准备在 ActiveNN 出问题的时候接管其功能和作用,变成 ActiveNN,这称为 NameNode 的“倒换(failover)”。NameNode 的倒换有两种,一种是手动,一种是自动。前者是由系统管理员通过键盘命令启动的,后者则由系统自行启动而无须人为干预。

作为对于 HDFS 的 HA 管理手段,Hadoop 提供了一个类似于 FsShell 那样的命令行工具 DFSHAAdmin,这个类是对抽象类 HAAdmin 的扩充。HAAdmin 意为“HA 管理”。

先看通过 HAAdmin 命令行的手动倒换,注意 HAAdmin 是抽象类,所以下面凡是 HAAdmin(在我们这个情景中)实际上都是指 DFSHAAdmin:

```

HAdmin.runCmd(String[] argv)
> String cmd = argv[0]
> ...
> CommandLine cmdLine = parseOpts(cmd, opts, argv)
> if (" - transitionToActive".equals(cmd)) return transitionToActive(cmdLine)
> else if (" - transitionToStandby".equals(cmd)) return transitionToStandby(cmdLine)
> else if (" - failover".equals(cmd)) return failover(cmdLine)
> else if (" - getServiceState".equals(cmd)) return getServiceState(cmdLine)
> else if (" - checkHealth".equals(cmd)) return checkHealth(cmdLine)
> else if (" - help".equals(cmd)) return help(argv)

```

以 failover 为例,命令的使用方法是:“hdfs haadmin -failover”。与 failover 相似的 HA 管理命令还有 transitionToActive 和 transitionToStandby,但是 Hadoop 的文档中明确劝告尽量不要用这两条,而应该用 failover。不过,作为代码的阅读分析,从 transitionToActive 开始倒也不坏。这个命令行的格式是“hdfs haadmin-transitionToActive nnid”,这里 nnid 就是原来用作 StandbyNN 的那个节点的 ID。对于“-transitionToActive”,这里实际调用的是 HAdmin 的 transitionToActive()。HAdmin 是个抽象类,但扩充了 HAdmin 类的 DFSHAdmin 并未提供自己的 transitionToActive():

```

[HAdmin.runCmd() > HAdmin.transitionToActive()]

HAdmin.transitionToActive(final CommandLine cmd)
> String[] argv = cmd.getArgs()
> HAdminServiceTarget target = resolveTarget(argv[0])
    == DFSHAdmin.resolveTarget(String nnId)
>> conf = (HdfsConfiguration) getConf()
>> return new NNHAdminServiceTarget(conf, nameserviceId, nnId)
    //创建 NNHAdminServiceTarget 对象,代表着作为目标的 StandbyNN
>>> targetConf = new HdfsConfiguration(conf)
>>> NameNode.initializeGenericKeys(targetConf, nsId, nnId)
>>> String serviceAddr = DFSUtil.getNameNodeServiceAddr(targetConf, nsId, nnId)
>>> this.addr = NetUtils.createSocketAddr(serviceAddr, NameNode.DEFAULT_PORT)
>>> this.autoFailoverEnabled = targetConf.getBoolean(
    DFSConfigKeys.DFS_HA_AUTO_FAILOVER_ENABLED_KEY,
    DFSConfigKeys.DFS_HA_AUTO_FAILOVER_ENABLED_DEFAULT)
    //从配置中获取是否开启自动倒换机制
>>> if (autoFailoverEnabled) {    //如果开启
>>>+ int port = DFSZKFailoverController.getZkfcPort(targetConf)
>>>+ if (port != 0) setZkfcPort(port)
>>> }
>>> this.fencer = NodeFencer.create(targetConf, DFSConfigKeys.DFS_HA_FENCE_METHODS_KEY)

```



```

>>> this.nnId = nnId
>>> this.nsId = nsId
> HAuthServiceProtocol proto = target.getProxy(getConf(), 0)
    == HAuthServiceTarget.getProxy(Configuration conf, int timeoutMs)
    //建立与 target,即 StandbyNN 的 RPC 连接,并创建其 proxy
> req = createReqInfo() //创建状态迁移请求
>> new StateChangeRequestInfo(requestSource)
> HAuthServiceProtocolHelper.transitionToActive(proto, req) //发出 RPC 请求
    == HAuthServiceProtocolHelper.transitionToActive(HAuthServiceProtocol svc,
    StateChangeRequestInfo reqInfo)
>> svc.transitionToActive(reqInfo) //对目标节点即 StandbyNN 的 RPC 调用

```

对 StandbyNN 发出 RPC 调用后,经过 RPC 层的传递,到了 StandbyNN 节点上。还是一样,首先到达 NameNodeRpcServer,进入其 transitionToActive():

```

NameNodeRpcServer.transitionToActive(StateChangeRequestInfo req)
> nn.checkHaStateChange(req)
> nn.transitionToActive() == NameNode.transitionToActive()
>> namesystem.checkSuperuserPrivilege()
>> state.setState(haContext, ACTIVE_STATE)
    == StandbyState.setState(haContext, ACTIVE_STATE) //设置 NN 节点的状态
>>> if (s == NameNode.ACTIVE_STATE) { //如果新的状态是 ACTIVE_STATE
>>>+ setStateInternal(context, s) == HActiveState.setStateInternal(HAContext context, HActiveState s)
>>>+ prepareToExitState(context)
>>>+ s.prepareToEnterState(context)
>>>+ exitState(context) //离开老状态
>>>+ context.setState(s)
>>>+ s.enterState(context) == ActiveState.enterState(HAContext context) //进入新状态
>>>+>> context.startActiveServices()
>>>+ return
>>> }
>>> super.setState(context, s)

```

别的就不多说了,关键在于对 ActiveState.enterState()的调用。回头看一下 NameNode 的构造方法,StandbyNN 与 ActiveNN 在那里就是通过这个 enterState()分道扬镳的。那以后的流程,这里就不再赘述了。可想而知,那以后会执行前述的 startActiveServices()。

相比之下,“-transitionToStandby”在 ActiveNN 上所引起的操作则是:

```

NameNodeRpcServer.transitionToStandby()
> namesystem.checkSuperuserPrivilege()
> state.setState(haContext, STANDBY_STATE)
    == ActiveState.setState(haContext, STANDBY_STATE)
>> ...

```

```

>>> HATState.setStateInternal(HAContext context, HATState s)
>>> ...
>>>> StandbyState.enterState(HAContext context)
>>>>> context.startStandbyServices()

```

最后还是 StandbyState.enterState()。同样,那以后的流程就无须赘述了。

显然,实际操作时应先使 ActiveNN 转入 Standby,然后再使 StandbyNN 转入 Active,这样才能保证不发生冲突。然而靠人的操作来保证这一点是靠不住的。

明白了这些,我们再看另一条命令 failover:

```
[HAAAdmin.runCmd() > failover(cmdLine)]
```

```
HAAAdmin.failover(cmdLine)
```

```

> boolean forceFence = cmd.hasOption(FORCEFENCE)
> boolean forceActive = cmd.hasOption(FORCEACTIVE)
> String[] args = cmd.getArgs()
> HATServiceTarget fromNode = resolveTarget(args[0]) //原先的 ActiveNN
> HATServiceTarget toNode = resolveTarget(args[1])    //原先的 StandbyNN
> Preconditions.checkNotNull(
    fromNode.isAutoFailoverEnabled() == toNode.isAutoFailoverEnabled(),
    "Inconsistent auto-failover configs between %s and %s!", fromNode, toNode)
    //双方对于自动倒换机制的启用与否应该一致
> if (fromNode.isAutoFailoverEnabled()) { //如果启用了自动倒换
>+ return gracefulFailoverThroughZKFCs(toNode) //通过倒换控制器 zkfc 实现自动倒换
    //注意参数为 toNode,这是原先的 StandbyNN
> }
> //继续完成命令 failover,手动倒换:
> FailoverController fc = new FailoverController(getConf(), requestSource)
> fc.failover(fromNode, toNode, forceFence, forceActive)
== FailoverController.Failover(fromNode, toNode, forceFence, forceActive) //完成倒换
    //调用参数 forceFence 和 forceActive 均来自命令行选项
> out.println("Failover from " + args[0] + " to " + args[1] + " successful")
> return 0

```

这里有两种情景:第一种是 ActiveNN 和 StandbyNN 都启用了自动倒换机制,这时候的倒换受一个 ZooKeeper 服务器的控制,我们暂时先搁一下,后面再回头看这种情景;第二种是双方并未一致同意启用自动倒换,这时候 HAAAdmin 这一方要创建一个倒换控制器,即 FailoverController 对象,由这个倒换控制器 fc 实施倒换。我们先看后面这种情景。

倒换控制器 FailoverController 的数据结构部分很小,所定义的方法函数也没几个,我们可以直接就看它的 failover()方法:

```
[HAAAdmin.runCmd() > HAAAdmin.failover() > FailoverController.failover()]
```

```

FailoverController.failover(HAServiceTarget fromSvc, HAServiceTarget toSvc,
                             boolean forceFence, boolean forceActive)
> Preconditions.checkNotNull(fromSvc.getFencer(), "failover requires a fencer")
> preFailoverChecks(fromSvc, toSvc, forceActive)
> boolean tryFence = true
    //Try to make fromSvc standby,先让 ActiveNN 转入 Standby
> tryGracefulFence(fromSvc)
>> proxy = svc.getProxy(gracefulFenceConf, gracefulFenceTimeout)
    //与 ActiveNN 建立连接并创建其 proxy
>> proxy.transitionToStandby(createReqInfo())
    //先对 ActiveNN 进行 RPC 调用,使其转入 Standby 模式
>> return true
> tryFence = forceFence
> // Fence fromSvc if it's required or forced by the user
> if (tryFence) {
>+ fromSvc.getFencer().fence(fromSvc) == NodeFencer.fence(fromSvc) //隔离操作
> }

    // Try to make toSvc active,然后让(原先的)StandbyNN 转入 Active
> proxy = toSvc.getProxy(conf, rpcTimeoutToNewActive)
> HAServiceProtocolHelper.transitionToActive(proxy, createReqInfo())
    //再对原先的 StandbyNN 进行 RPC 调用,使其进入 Active 模式
> if (failed) { //要是在这一步过程中发生异常而失败,就得倒换回来,并发起异常
>+ msg = "Unable to failover to " + toSvc
>+ failover(toSvc, fromSvc, true, true) //failback,倒换回来,恢复原先的 ActiveNN
>+ FailoverFailedException(msg, cause) //发起倒换异常
> }

```

可见,这只是以程序来保证先让 ActiveNN 转入 Standby,然后再让 StandbyNN 转入 Active;如果第一步成功而第二步失败,就得倒换回去,让原先的 ActiveNN 恢复 Active,如此而已。这里的 transitionToStandby()和 transitionToActive()都是我们在前面就见过的。唯一的不同之处,是两步之间有个 fence()操作。

这里分别代表着 ActiveNN 和 StandbyNN 的是两个 HAServiceTarget 类对象,而 HAServiceTarget 内部有个成分 fencer,是个 NodeFencer 类对象,是在创建 HAServiceTarget 对象的时候一起创建的,这里 fromSvc.getFencer()所返回的就是代表着 ActiveNN 的那个 HAServiceTarget 对象的 fencer。然后调用其 fence()函数,这就是 NodeFencer.fence(fromSvc)。

我们知道,fence 是篱笆、围墙、隔离的意思,这里就有把原先的 ActiveNN 隔离一下的意思。这是什么意思呢?在同一个 Hadoop 集群中,如果不考虑联邦模式的话,我们只允许有一个 Active 的 NameNode;即使在联邦模式下,在同一个 Namespace 中也只允许有一个 Active 的 NameNode。现在我们要让原来 Active 的 NN 退出 Active,而让原来 Standby 的 NN 变成 Active,但是谁跑得快呢?这并不单纯取决于两边的机器速度和程序长度,还取决于两个节点

上宿主操作系统的调度,这里面有很多偶然因素,是我们无法控制的。如果后者跑得快,那就会有个时间窗口,在这个窗口中就有两个 ActiveNN 并存,这就会“打架”。要解决这个问题,最好的办法当然是让后者等待一下,确认前者已经退出 Active 才让后者进入 Active。即使在一短暂的时间中没有 ActiveNN,也比同时有两个 ActiveNN 要好。可是要做到这一点也并不容易,如果是在同一进程中,那我们可以设置一个标志变量,可以加锁,但是这不仅不在同一进程中,还不在于同一机器上。怎么办呢?我们可以安排一个或几个“观察员”,仅在至少有一个观察员确认已经看到那个 NameNode 已经退出 Active,或者干脆已经不复存在后,才让后者进入 Active。这样的机制,就称为 fence,观察员则就是 Fencer,这是某种作为进程或线程运行的应用程序,实际起作用的是其提供的具体观察方法。

Hadoop 的代码中有个字符串常数 DFS_HA_FENCE_METHODS_KEY:

```
String DFS_HA_FENCE_METHODS_KEY = "dfs.ha.fencing.methods"
```

用户可以在配置块 conf 中配置项“dfs.ha.fencing.methods”下寻找关于 Fencer 的设置,这可以是一串以逗号分隔的方法名。Hadoop 提供了两种 Fencer,一是 shell,一是 sshfence。其实有这两个就足够了,因为例如 ssh 就可以用来执行一个脚本,那已经是要让它有多复杂就可以有多复杂了。

这里调用的 NodeFencer.fence(),其代码摘要是这样:

```
[HAdmin.runCmd() > failover() > FailoverController.failover() > NodeFencer.fence()]
```

```
NodeFencer.fence(HAServiceTarget fromSvc)
> LOG.info("==== Beginning Service Fencing Process...====")
> for (FenceMethodWithArg method : methods) { //一个个试过来
>+ success = method.method.tryFence(fromSvc, method.arg)
    == ShellCommandFencer.tryFence(HAServiceTarget target, String cmd)
>+> if (!Shell.WINDOWS) { //操作系统为 Linux:
>+>+ builder = new ProcessBuilder("bash", "-e", "-c", cmd)
>+> } else { //操作系统为 Windows:
>+>+ builder = new ProcessBuilder("cmd.exe", "/c", cmd)
>+> }
>+> setConfAsEnvVars(builder.environment())
>+> addTargetInfoAsEnvVars(target, builder.environment())
>+> Process p = builder.start() //启动该命令行,创建子进程
>+> p.getOutputStream().close() //关闭其输出流
>+> ...
>+>+ rc = p.waitFor() //等待子进程结束
>+> return rc == 0
>+ if(success) {
>+> LOG.info("==== Fencing successful by method " + method + "====")
>+> return true //只要有一个成功就返回 true
>+ }
```

```
> }  
> LOG.error("Unable to fence service by any configured method.")  
> return false //全部失败就返回 false
```

不过我们现在并不关心这个,只要知道有 fence 这么回事就行了。

16.4 Zookeeper 与自动倒换

前面所讲都是手动倒换,现在我们回头看前述的另一种情景,即 ActiveNN 和 StandbyNN 都启用了自动倒换机制时的倒换。但请注意,这只是在二者都启用了自动倒换条件下执行 failover 命令时的流程, failover 这条命令还是手动输入的;而真正意义上的“自动倒换”,则要连倒换过程的触发启动也是自动的才算,所以这只是自动倒换的一部分。不过,虽然只是一部分,这个过程也要依靠 ZooKeeper 服务器(ZooKeeper Server)所提供的服务。至于倒换过程的自动触发,既然有了 ZooKeeper 服务器,则自然也就有了。

顾名思义,ZooKeeper 就像是“动物园”的管理员,像 ActiveNN、StandbyNN 那样存在竞争关系的对象就像是“动物”。不过这个比喻其实不是很贴切,因为 ZooKeeper 对于 ActiveNN/StandbyNN 之类并没有管理和“饲养”的作用,而只是为它们提供了竞争和协调的手段。

ZooKeeper 也是个开源软件,编译以后产生的程序包就是 org.apache.zookeeper,凡要引用 ZooKeeper 的 Java 程序必须从这个包中导入。此外,如果采用 ZooKeeper,就得在集群中部署 ZooKeeper“服务器”,由它提供竞争手段。

对于 ZooKeeper 的作用,我们可以这样来理解:在同一进程内部的两个线程之间,同步与互斥可以通过加锁来实现,而加锁的基础是两个线程有可以共享的变量,即存储单元,由于同一进程中的线程共享同一内存空间,所以共享变量不是问题。也有人通过自行定义和使用的标志位、标志变量来实现同步与互斥,其实是同样的道理,关键是共享变量。我们把视野扩大一圈,看同一机器上不同进程间的同步与互斥。进程的(用户)内存空间是独立的,不同进程间一般而言没有可以共享的变量,所以进程间的同步与互斥只能靠操作系统(以系统调用的方式)提供手段,因为在内核中、在系统空间还是可以有共享变量的,毕竟双方都在同一台机器上。当然,现代操作系统还提供进程间共享内存的机制,如果两个进程有共享内存,那就又可以依赖用户空间的共享变量来实现进程间的同步与互斥了。然而,再把视野扩大一圈,看集群中的两台不同机器之间,这时候就没有共享变量、没有双方 CPU 都可以寻址访问的存储单元了。怎么办呢?办法就是在集群内的某个节点上安上一个 Server,让所有节点都可以通过网络通信共享这个 Server 中的变量。如此,就可以用这个 Server 上的变量来实现加锁以及类似于 P/V 操作那样的机制。

Zookeeper 就是按这个思路设计和实现的,Zookeeper 的 Server 就是这样的服务器。Zookeeper 服务器中提供了一个形似文件目录的树形结构,树上的节点称为 znode。树上的叶节点都是用于同步、互斥的数据节点,即共享变量,每个叶节点都可带上少量数据,包括该节点的所有人和操作序号等信息;中间节点则为目录节点。如果集群中有 A、B 两个进程要竞争进行某项操作(相当于进入临界区),就可按预先约定的相同路径请求在服务器上创建一个叶节点,这些操作请求在服务器上会被排队得到执行,因而总会有个先后。于是,先来者,假定为

A, 便得以创建节点而成功返回, 就像加上了锁; 后来者, 假定为 B, 则因为同名(同路径)节点已经存在而失败返回。然后失败者 B 可以对此节点加“关注(Watch)”; 服务器在 A 删除该节点(相当于解锁)时, 或者服务器因失去与 A 的连接而删除该节点时, 会对关注者发出通知, 让 B 可以再来创建这个节点(相当于加锁)。所以, znode 有两种, 一种是永久性的, 例如目录节点就是永久性的(除非特别加以删除); 另一种则是“短生(ephemeral)”性的, 如数据节点, 即叶节点, 其生存期最长(如果不主动加以删除)也只能延续到服务器与其所有者失去联系时为止。一旦服务器与其所有者失去联系, 就会自动把这个节点删除, 并向关注者发出通知。或者, 如果所有者主动删除一个节点, 服务器也会向关注者发出通知。

显然, 由 Zookeeper 服务器提供的这种跨节点的进程同步/互斥机制正可用于节点的热备份自动倒换, 包括 ActiveNN 与 StandbyNN 之间的自动倒换, 因为 NameNode 之进入 Active 状态就好比进入临界区一样。具体的方法就是让两个 NameNode 都以相同的路径企图在 Zookeeper 服务器上创建一个节点, 创建成功的就是 ActiveNN, 失败的就是 StandbyNN。运行中, 如果 ActiveNN 与 Zookeeper 服务器失去了联系, 服务器便会删除那个节点, 并向 StandbyNN 发出通知, 让其前来创建同名的节点以求变成 ActiveNN。至于原来的 ActiveNN, 则当它重启或恢复的时候会从 Zookeeper 服务器获悉已经失去 Active 的资格而转入 Standby。

但是这里又有个问题, 万一 Zookeeper 服务器失效怎么办, 这岂非又是单点故障? 所以, Zookeeper 服务器既可以是单台的, 也可以(更应该)是个由多台服务器构成的“议会(Quorum)”。可是在一个 Quorum 中的多台服务器也得有个主(Master)备之分, 那不是又回到 Active/Standby 的问题上来了吗? ActiveNN 与 StandbyNN 靠 Zookeeper 服务器协调, 然而 Zookeeper 服务器的 Quorum 又靠谁来协调? 显然, 不能在 Zookeeper 服务器的 Quorum 上面又堆上一个 Zookeeper 服务器。所以, 在 Zookeeper 服务器的 Quorum 中采用的是投票选举的策略, 让 Quorum 中的那些 server 通过投票选出一个 Leader, 来出面代表整个 Quorum 提供服务, 万一这个 Leader 在运行中失败就再次投票选举。注意, 这种投票选举只是为了在构成 Quorum 的多个 Zookeeper 服务器中选出一个 Leader, 而不是在 Zookeeper 的用户即借助 Zookeeper 协调互斥的那些节点之间的选举。Zookeeper 的用户之间, 例如 ActiveNN 和 StandbyNN 之间, 是没有投票选举的, 那里有的只是类似于加锁和 P/V 操作那样的互斥和抢占。不过, 话虽如此, 实际上人们往往还是把这也看成是“选举(Election)”, Hadoop 的源码中就有个 ActiveStandbyElector 类, 给人印象仿佛 Active 是选出来的。

对于 Hadoop 而言, ZooKeeper 服务器是个第三方软件, 其代码不在 Hadoop 中, 但是如果开通 HA 实现自动倒换就要部署并启用(单个或多个) ZooKeeper 服务器。当然, 还得部署和启用 zkfc, 即 ZKFailoverController。其中 zkfc 是 Hadoop 这一边 ActiveNN 和 StandbyNN 双方通过 ZooKeeper 服务器进行竞争和互斥并实施倒换的控制器。实际上这也不仅仅是用在两个 NameNode 之间, 凡是需要有 Active/Standby 热备的设施都可以用; 例如 Hadoop 上的数据库 HBase, 就也采用了这种机制。

显然, zkfc 必须与 ZooKeeper 服务器通信, 并按 ZooKeeper 的规程操作。为此, Hadoop 从 org.apache.zookeeper 包中导入其提供的 ZooKeeper 类, 作为 Hadoop 通向 ZooKeeper 服务器的接口模块, 对于 Hadoop 而言这就代表着 ZooKeeper, 相当于 ZooKeeper 服务器的 proxy; 对于 ZooKeeper 服务器, 则这是其客户端, 代表着 ZooKeeper 服务的使用者。

为实现自动倒换,ActiveNN 和 StandbyNN 所在的节点上都要有个倒换控制器 zkfc,即运行在独立 Java 虚拟机上的 ZKFailoverController 类对象。

ZKFailoverController 是个抽象类,DFSZKFailoverController 则是对这个抽象类的扩充,显然这是用于 HDFS 的 zkfc。DFSZKFailoverController 提供 main()函数,编译之后名为 zkfc,在一个独立的 JVM 上运行。如果采用则凡是运行着 NameNode 的节点上都应该有个运行着 zkfc 的 JVM 进程。

回头看前面 HAAdmin.failover()的代码摘要,如果启用了自动倒换,那么此时调用的是 gracefulFailoverThroughZKFCs():

```
[HAAdmin.runCmd() > HAAdmin.failover(cmdLine) > gracefulFailoverThroughZKFCs()]
```

```
gracefulFailoverThroughZKFCs(HAServiceTarget toNode)
```

```
> ZKFCProtocol proxy = toNode.getZKFCProxy(getConf(), timeout)
```

```
> proxy.gracefulFailover() //对 zkfc 的 RPC 调用
```

注意,这个函数名后面有个 s,表示 ZKFC 是复数,不止一个。但是这里的参数为 toNode,这是原先处于 StandbyNN、现在要倒换成为 ActiveNN 的那个节点;所以这里的 proxy 代表着那个节点上的 zkfc。

跟 NameNode 上有个 NameNodeRpcServer 一样,ZKFailoverController 上也有个 ZKFCRpcServer,这个 RPC 就成为对其 gracefulFailover()函数的调用:

```
[HAAdmin.runCmd() > failover(cmdLine) > gracefulFailoverThroughZKFCs()]
```

```
=> ZKFCRpcServer.gracefulFailover()]
```

```
ZKFCRpcServer.gracefulFailover()
```

```
> zkfc.checkRpcAdminAccess()
```

```
> zkfc.gracefulFailoverToYou() == ZKFailoverController.gracefulFailoverToYou()
```

```
>>> UserGroupInformation.getLoginUser().doAs(new PrivilegedExceptionAction<Void>())
```

```
] run()
```

```
> doGracefulFailover()
```

显然,这里先查验权限,然后就以请求者的身份执行 doGracefulFailover(),这一步操作分成五个阶段:

```
[ZKFCRpcServer.gracefulFailover() > ZKFailoverController.gracefulFailoverToYou()]
```

```
> doGracefulFailover()]
```

```
ZKFailoverController.doGracefulFailover()
```

```
> timeout = FailoverController.getGracefulFenceTimeout(conf) * 2
```

```
> // Phase 1: pre-flight checks, ensure that the local node is healthy,
```

```
and thus a candidate for failover.
```

```
> checkEligibleForFailover()
```

```

>>> s = this.getLastHealthState()
>>> if (s!= State.SERVICE_HEALTHY) ServiceFailedException(
        localTarget + " is not currently healthy. " + "Cannot be failover target")
> // Phase 2: determine old/current active node. Check that we're not ourselves active, etc.
> HAServiceTarget oldActive = getCurrentActive() //获知倒换前的 ActiveNN 所在节点
>>> byte[] activeData = elector.getActiveData() == ActiveStandbyElector.getActiveData()
>>>> if (zkClient == null) createConnection()
>>>>> if (zkClient != null) {
>>>>>+ zkClient.close()
>>>>>+ zkClient = null
>>>>> }
>>>>> zkClient = getNewZooKeeper() //对 ZooKeeper 服务器而言的客户端 zkClient
>>>>>> watcher = new WatcherWithClientRef()
>>>>>> zk = new ZooKeeper(zkHostPort, zkSessionTimeout, watcher)
//创建 ZooKeeper 对象,通过它连接到 ZooKeeper 服务器
>>>>>> watcher.setZooKeeperRef(zk)
>>>>>> watcher.waitForZKConnectionEvent(zkSessionTimeout)
>>>>>> for (ZKAuthInfo auth : zkAuthInfo) {
>>>>>>+ zk.addAuthInfo(auth.getScheme(), auth.getAuth())
>>>>>> }
>>>>>> return zk
>>> stat = new Stat()
>>> getDataWithRetries(zkLockFilePath, false, stat)
//从 ZooKeeper 获取数据,以确定谁是 ActiveNN,zkLockFilePath 就是路径
>>>>> action = new ZKAction<byte[]>()
        ] run()
        > zkClient.getData(path, watch, stat)
>>>>> return zkDoWithRetries(action)
>>>>>> while (true) {
>>>>>>+ try {
>>>>>>++ return action.run()
>>>>>>++> zkClient.getData(path, watch, stat) == ZooKeeper.getData(path, watch, stat)
>>>>>>+} catch (KeeperException ke) {
>>>>>>++ if (shouldRetry(ke.code()) && ++ retry < maxRetryNum) continue
>>>>>>++ throw ke
>>>>>>+ }
>>>>>> }
>>> HAServiceTarget oldActive = dataToTarget(activeData)
>>> return oldActive
> if (oldActive == null) { //如果当前没有 ActiveNN

```

```

>+ // No node is currently active. So, if we aren't already active ourselves by means of
>+ // a normal election, then there's probably something preventing us from becoming active.
>+ ServiceFailedException("No other node is currently active.")
> }
> if (oldActive.getAddress().equals(localTarget.getAddress())) { //如果本身已是 ActiveNN
>+ LOG.info("Local node " + localTarget + " is already active. " +
        "No need to failover. Returning success.")
>+ return
> }
> // Phase 3: ask the old active to yield from the election.
> LOG.info("Asking " + oldActive + " to cede its active state for " + timeout + "ms")
> ZKFCProtocol oldZkfc = oldActive.getZKFCProxy(conf, timeout) //建立与对方的通信
> oldZkfc.cedeActive(timeout) //要求对方让出活跃角色,并在 timeout 时间段内不参选
> // Phase 4: wait for the normal election to make the local node active.
> ActiveAttemptRecord attempt = waitForActiveAttempt(timeout + 60000)
        //等待本节点被调用 becomeActive()而得到 ActiveAttemptRecord,或者超时
> if (attempt == null) { //没有得到 ActiveAttemptRecord,超时
>+ // We didn't even make an attempt to become active.
>+ if (lastHealthState != State.SERVICE_HEALTHY) {
>++ ServiceFailedException("Unable to become active. " +
        "Service became unhealthy while trying to failover.")
>+ }
>+ ServiceFailedException("Unable to become active. " +
        "Local node did not get an opportunity to do so from ZooKeeper, " +
        "or the local node took too long to transition to active.")
> }
        //本节点得到了 becomeActive()调用,得到了一个 ActiveAttemptRecord 对象即 attempt
        //此时的本节点已经变成活跃主节点
> // Phase 5. At this point, we made some attempt to become active.
        So we can tell the old active to rejoin if it wants.
> // This allows a quick fail - back if we immediately crash.
> oldZkfc.cedeActive(-1) //允许对方立即参选,此时本节点已是 Active
> if (attempt.succeeded) {
>+ LOG.info("Successfully became active. " + attempt.status)
> } else {
>+ msg = "Failed to become active. " + attempt.status
>+ ServiceFailedException(msg)
> }

```

摘要中已经加了注释,读者对于宏观的过程应已不难理解,这里再对几个关键点作些说明。首先是谁会给这个节点发来它想要的 becomeActive(),这个问题后面还要详述,这里先

暂搁一下。

如前所述,此时 ZooKeeper 服务器上路径为 zkLockFilePath 的 znode 是由 ActiveNN 所持有的,我们不能指望 ActiveNN 忽然就与 ZooKeeper 服务器断掉了联系,而只能与其友情协商,请求它退让一下,这就是这里的 oldZkfc.cedeActive(timeout)。

ActiveNN 所在节点上的 zkfc(即 oldZkfc)收到要求其退让的报文后,其 PB 层即调用 ZKFCRpcServer 的 cedeActive():

```
ZKFCRpcServer.cedeActive(int millisToCede) //原来的 Active 节点被要求 cedeActive()
> zkfc.checkRpcAdminAccess()
> zkfc.cedeActive(millisToCede) == ZKFailoverController.cedeActive(final int millisToCede)
>> UserGroupInformation.getLoginUser().doAs(new PrivilegedExceptionAction<Void>()
    ] run()
    > doCedeActive(millisToCede)
    >> timeout = FailoverController.getGracefulFenceTimeout(conf)
    >> if (millisToCede <= 0) {
    >>+ delayJoiningUntilNanotime = 0
    >>+ recheckElectability()
    >>+ return
    >> }
    >> localTarget.getProxy(conf, timeout).transitionToStandby(createReqInfo())
    //RPC,让该节点上的 NN 转入 Standby
    >> LOG.info("Successfully ensured local node is in standby mode")
    >> elector.quitElection(needFence) == ActiveStandbyElector.quitElection(needFence)
    //当前 ActiveNN 退出竞选,意味着删掉路径为 zkLockFilePath 的 znode,让出 Active
    >> serviceState = HAServiceState.INITIALIZING
    >> recheckElectability()
    > return null
```

这样,原先 Active 的 NameNode 通过 quitElection()退出了竞争,就会开始新一轮的竞争。在我们这个情景中,一共才两个 NameNode,其中一个已经退出,双方开始了新一轮竞争。

注意,退出 Active 的这一方在其 cedeActive()中最终还是调用 zkfc.cedeActive(),那就是 ZKFailoverController.cedeActive(),而想要变成 Active 的那一方,则自然也是调用这个函数。双方都是通过 recheckElectability()进入新一轮竞争,但是怎么保证这一次是后者必胜呢?

```
[ZKFCRpcServer.cedeActive() > ZKFailoverController.recheckElectability()]
//Check the current state of the service, and join the election if it should be in the election.
ZKFailoverController.recheckElectability()
> boolean healthy = lastHealthState == State.SERVICE_HEALTHY
> remainingDelay = delayJoiningUntilNanotime - System.nanoTime()
```

```

> if (remainingDelay > 0) {
>+ scheduleRecheck(remainingDelay)
>+ return //刚让出 Active 的 zkfc, 因 remainingDelay>0 而暂不参与竞争
> }
> switch (lastHealthState) {
> case SERVICE_HEALTHY:
>+ elector.joinElection(targetToData(localTarget)) //原先的 Standby 立即参与竞争, 必胜
>+ if (quitElectionOnBadState) quitElectionOnBadState = false
> case INITIALIZING:
>+ elector.quitElection(false)
>+ serviceState = HAServiceState.INITIALIZING
> case SERVICE_UNHEALTHY:
> case SERVICE_NOT_RESPONDING:
>+ elector.quitElection(true)
>+ serviceState = HAServiceState.INITIALIZING
> }

```

我们在前面看到, 想要变成 Active 的这一方在通过 `cedeActive()` 要求对方退让时, 还要求它在 timeout 时间段内不参选, 这里的 `if (remainingDelay > 0)` 则保证了这一点。这样, 只要这个 timeout 足够长, 事实上它就不会来竞争, 所以原先的 Standby 这一方是独家竞争, 稳操胜券。于是, ZooKeeper 服务器就会给它发出当选的通知。当报文到达这个原来处于 Standby 地位的节点时, 它的 ZooKeeper 客户端就会回调这个节点上的 `ActiveStandbyElector` 所提供的回调函数 `processResult()`:

```

ActiveStandbyElector.processResult(int rc, String path, Object ctx, Stat stat)
> if (isStaleClient(ctx)) return
> assert wantToBeInElection : "Got a StatNode result after quitting election"
> Code code = Code.get(rc)
> if (isSuccess(code)) { // code == Code.OK
>+ if (stat.getEphemeralOwner() == zkClient.getSessionId()) {
>++ becomeActive() //变成 Active
>++> if (state == State.ACTIVE) return true //already active
>++> Stat oldBreadcrumbStat = fenceOldActive()
>++> writeBreadcrumbNode(oldBreadcrumbStat)
>++> LOG.debug("Becoming active for " + this)
>++> appClient.becomeActive() == ActiveStandbyElectorCallback.becomeActive()
>++>> ZKFailoverController.this.becomeActive() == ZKFailoverController.becomeActive()
>++>>> LOG.info("Trying to make " + localTarget + " active...")
>++>>> svc = localTarget.getProxy(conf, FailoverController.getRpcTimeoutToNewActive(conf))
>++>>> HAServiceProtocolHelper.transitionToActive(svc, createReqInfo())
>++>>> LOG.info("Successfully transitioned " + localTarget + " to active state")

```

```

> ++> > > serviceState = HAServiceState.ACTIVE
> ++> > > recordActiveAttempt(new ActiveAttemptRecord(true, msg))
> ++> state = State.ACTIVE
> ++> return true
> ++ if(!becomeActive()) reJoinElectionAfterFailureToBecomeActive()
> + } else {
> ++ becomeStandby() //不是变成 Active,就是变成 Standby
> + }
> + return
> } //end if(isSuccess(code))
> if (isNodeDoesNotExist(code)) {
> + enterNeutralMode()
> + joinElectionInternal()
> + return
> }
> String errorMessage = "Received stat error from Zookeeper. code:" + code.toString()
> LOG.debug(errorMessage)
> if (shouldRetry(code)) {
> + if (statRetryCount < maxRetryNum) {
> ++ ++ statRetryCount
> ++ monitorLockNodeAsync()
> ++ return
> + } else if (isSessionExpired(code)) {
> ++ LOG.warn("Lock monitoring failed because session was lost")
> ++ return
> + }
> }
> fatalError(errorMessage)

```

至此,在 ZooKeeper 控制器的控制下,两个 NN 节点的角色就完成了自动倒换。

回到前面的 doGracefulFailover(),那里又调用了一次 oldZkfc.cedeActive(),但是这次的参数是-1,让对方立即参加竞争。然而此时已经有 Active 了,所以对方只是变成了 Standby,担负起作为 StandbyNN 需要承担的义务。

至于倒换的自动触发,其实更为简单。当前的 Active 一旦发生故障失去了与 ZooKeeper 服务器之间的联系,ZooKeeper 服务器就会自动删除由其创建的 znode 并发出通知,此时前来竞争的就是原先的 Standby 一家。

16.5 YARN 的 HA 机制

在 Hadoop 老一点(2.4 版之前)的版本中,资源管理器(Resource Manager, RM)仍是个

单点失效会引起全局失效的薄弱环节,后来才加入了 Active/Standby 即“热备”的机制。有了这种热备机制以后,在任何一个时间点上集群中都只有一个节点起着“当班” RM 的作用,处于 Active 状态;但是却可以另有一个或多个节点起着候补 RM 的作用,处于 Standby 状态。同样,这种角色和状态的倒换可以通过控制台命令手动,也可以自动。自动倒换是一个可配置的选项,但是选用了自动便不再允许手动,我们在这里集中关注自动。

RM 的热备机制也是可选项,需要在配置中加以启用。

Hadoop 的源文件 YarnConfiguration.java 中定义了许多字符串,其中的几个是这样的:

```
String RM_PREFIX = "yarn.resourcemanager."
String RM_HA_PREFIX = RM_PREFIX + "ha."
String RM_HA_ENABLED = RM_HA_PREFIX + "enabled"
```

这样, RM_HA_ENABLED 就是“yarn.resourcemanager.ha.enabled”,在 YARN 的配置文件 yarn-default.xml 中就有这么一条属性,缺省的设置是 false。要启用 RM 的容错机制就得将此选项改成 true。

```
class RMAAdminCLI extends HAAdmin {}
] run(String[] args)
> ...
> isHAEnabled = yarnConf.getBoolean(YarnConfiguration.RM_HA_ENABLED,
                                     YarnConfiguration.DEFAULT_RM_HA_ENABLED)
> ...
```

如果 YARN 子系统启用了 HA 机制,那么 ResourceManager 对象在创建之初都是按 Standby 模式初始化的,直至受到来自 ZooKeeper 的 becomeActive() 调用之后才会通过 transitionToActive() 转入 Active 模式,这时候才会调用 ResourceManager.startActiveServices(),而后的操作就是调用 activeServices.start()。这里 activeServices 是 ResourceManager 内部的一个 RMAActiveServices 对象。

RMAActiveServices 类是对 CompositeService 类的扩充,后者又是对抽象类 AbstractService 的扩充;而 ResourceManager 中的 activeServices 就是个 RMAActiveServices 对象,这我们在本书第三章中讲 ResourceManager 的那一节里看到了,只不过那时候我们假定不启用 HA,所以 RM 并不需要有 becomeActive() 就进入了 Active 模式。现在,在启用了 HA 的条件下,情况就不一样了。

```
[ActiveStandbyElector.ActiveStandbyElectorCallback.becomeActive()
> AdminService.transitionToActive()>ResourceManager.startActiveServices()
> RMAActiveServices.start()]
```

```
RMAActiveServices.start()
== AbstractService.start() // RMAActiveServices 和 CompositeService 均未定义 start()
> serviceStart() == RMAActiveServices.serviceStart()
>> RMStateStore rmStore = rmContext.getStateStore()
>> rmStore.start() //启动 RMStateStore 对象 rmStore
```

```
>>> if(recoveryEnabled) {
>>>+ rmStore.checkVersion()
>>>+ RMState state = rmStore.loadState()
//如果启用了 RMStateStore 的恢复机制,则还要从 RMStateStore 加载
>>>+ recover(state)
>>> }
>>> super.serviceStart() == CompositeService.serviceStart()
>>>> List<Service> services = getServices()
>>>> for (Service service: services) {
>>>>+ service.start() //启动作为 ActiveRM 所需提供的各项服务
>>>> }
```

由此可见,与 StandbyRM 相比,ActiveRM 的特殊之处在于两个方面:一个方面是要提供作为 ActiveRM 所需提供的各项服务,所以这里要启动有关的各项服务。还有一个方面,则有关用来存储已提交作业状态的 RMStateStore;所以这里要通过 rmStore.start() 启动 RMStateStore 对象 rmStore。如果启用了 RMStateStore 的恢复机制,则还要从 RMStateStore 加载其内容。在某种意义上,RMStateStore 与 HDFS 的文件系统映像和 EditLog 有些相似。

同样,两个 RM 节点也可以在 ZooKeeper 控制器的控制下完成自动倒换,不过 RM 节点的倒换渗透在 RM 子系统的代码中,而并不专门使用一个独立的 zkfc。虽然并不使用独立的 zkfc,其工作原理与上述 HDFS 子系统中 NameNode 的倒换还是大同小异的,而且 ZooKeeper 的工作机理其实已经超出 Hadoop 的范畴,这里就不详述了,有兴趣或需要的读者可自行研究分析。

第17章

Hadoop 的安全机制

17.1 大数据集群的安全问题

一个系统,无论其为单机还是集群,如果把它比作一个大院,那么这个大院的安全问题主要就是两条:一是允许什么人进来,这是“准入”的问题;二是允许进来的人在院内做些什么,这是“准行”的问题。当然,还有个问题是可带走些什么,但是那其实只是“做些什么”的一部分。

允许什么人进来的问题,是通过“登录(Login)”解决的,关键在于录入的依据。这样的依据无非是两种:一是双方共知或共有的某种特征或秘密,例如口令;二是由获得认可的第三方颁发的证件。当然,也可以是二者的结合,总之就是要搞清这是什么人或者属于哪一类的人。也就是说,问题的核心在于搞清要求进入者的身份。所以,解决准入问题的手段就是对于“用户”的身份认证(Authentication)。

允许做什么的问题,就更复杂了。虽然都被允许进入大院了,但是允许各种人在院内可以采取的行为却有不同,因为各人对院内财物的所有权、能力、忠诚度等皆有不同,更何况百密一疏也许有人混了进来。所以,对获准进入系统者的行为应该有明确的授权和限制,并在各种行为发生之前进行权限的检验,这样的机制称为“权限管理(Authorization)”。

任何一个系统,出于安全的考虑,都应该有身份认证和行为授权这么两种功能和机制。一般在操作系统这一层上就有用以解决身份验证的 Login 机制,通过用户名和口令(Password)的比对解决身份认证的问题。这对于有限几个用户的小众在本地使用还比较简单;但是如果是为大众所使用,特别是通过网络的远程使用,那么身份认证的问题可以变得很复杂,因为口令特别是比较简单的口令是有可能被攻破的。不过要让每一台机器都具有很强的身份认证功能却是个不小的负担,所以有个办法就是把它委托、“外包”给一个第三方。当然这里面又有许多新的、具体的问题,但是人们已经基本解决了这些问题,Kerberos 就是一种广泛采用的第三方身份认证技术,许多服务器都采用了这种技术。

权限管理的问题更加复杂。Linux 的前身 Unix 从一开始就有区分“三种人”(文件主、组人、其他人),并划分“特权用户”和“普通用户”的文件访问权限机制。这种机制是以文件为中心的,可以为每个文件设置哪一些人可以对它做些什么,一般就归结为读、写、执行三种。由于 Unix 把设备也看成特殊文件,这种访问权限的设置和运用其实已经超出了一般意义上“文件”的范畴。事实上,到现在为止,大量安全问题的发生都是因为低级错误,即没有善用这种原始的文件访问权限机制所导致的。所以低级错误都是管理问题,高级错误才是技术问题。

不过,对于某些行业的应用来说,传统的那种只区分三种人的管理方式确实还是不够精细,所以后来才有了“访问控制名单(Access Control List)”即 ACL 的机制。这种机制也是以文件为中心,对于每个文件(和目录)都可以详细列举谁可以对它做些什么。显然,这只是对传统文件访问权限机制的扩充和细化。既然可以有以文件为中心的 ACL,那当然也可以有以用户为中心的 ACL,对每个用户都逐一列举可以做什么。而“做些什么”,当然也可以扩展到文件访问之外的其他活动,甚至在什么情况下可以做什么。所以,后来才有了 SELinux,即安全性增强了的 Linux;然而 SELinux 的使用相当复杂,不是一般人都能用得起来。

尽管操作系统这一层上已经提供了身份验证和授权的机制和手段,在具体的应用层面还是有怎么用好这些机制和手段的问题,并且也还有改进、补充、扩展的空间。Java 语言的安全机制 JAAS(Java Authentication and Authorization)就是这么来的。当然,JAAS 只适用于以 Java 语言编写的应用。JAAS 在其设计之初或许是针对单机环境的,但是实际上也为集群环境的安全机制提供了一个很好的基础,Hadoop 的安全机制就是建立在 JAAS 的基础之上。

当然,从单机到集群,安全性的问题变得更复杂了。集群的安全问题显然不会比单机简单,一个集群的安全程度也不太可能高于单机。比方说集群中各节点间的网络通信,比之单机中的进程间通信,就是个更薄弱的环节,因为网络中的通信更容易被监听、拦截和假冒。身份认证和行为授权这两个问题在 Hadoop 集群中也变得更复杂了,因为同一个用户的活动将分布在多个节点上,而同一个文件的内容也将分布在多个节点上。不过倘若与公共的广域网相比,则又要简单很多,因为毕竟每个 Hadoop 集群通常都集中在一起,是有统一管理和维护的。所以,如果能把住用户的准入这个关口,再把单机中的安全措施移植、推广到集群中,并考虑集群的特殊性加上一些补充,就有可能使 Hadoop 集群的安全性达到与单机相仿的水平。事实上,Hadoop 也正是从这么三个方面着手的。

在 Hadoop 集群中,一般不会对每个节点机都带上键盘和监视屏,也不会让主节点带上很多的键盘和监视屏,用户一般总是通过网络接入而并不现身于管理员面前,整个 Hadoop 集群在逻辑上就相当于一台或几台服务器。这样,既然允许用户远程接入,那么身份认证(Authentication)就是个问题。使用 password 的 Login 是最简单的身份验证,更高级的则采用 Kerberos 之类的第三方验证,所以 Hadoop 支持采用 Kerberos。

把单机上的安全措施移植、推广到集群,就比较复杂了。如前所说,在 Hadoop 集群上用户的活动可能遍及整个集群,怎样让用户随身携带其身份信息,怎样维持访问权限在所有相关节点上的一致性,这都是必须解决的问题。

由于是在地理和管理都比较集中的局域网环境中,网络通信的加密没有身份验证和授权那么关键,再说技术也都是现成的。而身份认证,只要不把口令设置得过于简单,并限制允许出错的次数,进而结合加密通信,应该说安全性就已经有了基本保障。即使 Kerberos,也是在企业应用环境(例如 Samba)中常用的技术。相比之下,对于访问和行为的控制,即“权限管理”就显得更为关键。

在 Hadoop 中,安全机制是作为选项提供的,Hadoop 默认运行于不开启安全机制的模式,如果要开启安全机制,就需要设定一些选项。这些选项,有些是在 Java 语言这一层的,有些是在 Hadoop 这一层的。在总体上开启安全机制的前提下,还可以就许多单项进行配置选择。例如对于身份认证就可以选择 Login、Kerberos、Token;对于行为的权限管理可以选择是否让

系统通过 `doAs()` 以具体用户的身份进行操作;对于资源的访问控制可以选择是仅仅使用传统的文件访问权限控制,还是也要采用 ACL;对于通信则可以选择是否加密。如此等等,不一而足。

也许是因为考虑到通常一开始只是部署在单机上,即便部署在集群上开始时也只是实验室中的小集群,一来在那些环境中安全不是问题,二来要降低试用的门槛,所以 Hadoop 源码包中的那些.xml 配置文件都是按最简单的方式设置的,那就是不启用安全机制,即使启用也把身份验证的方式设为 Simple,并且节点间的通信不加密。但是如果把 Hadoop 部署于较大并且使用情况比较复杂的集群,就应该考虑修改有关的配置文件,相关的选项分布在 `core-default.xml`、`hdfs-default.xml`、`hadoop-policy.xml`、`yarn-site.xml`、`mapred-site.xml`、`hdfs-site.xml`、`web.xml` 等配置文件中。有关配置项的名称各异,但只要以 security 为关键词搜一下就都可以找到。

在具体实现的层面,Hadoop 的安全机制是建立在 Java 的安全机制 JAAS 的基础之上。而 JAAS,虽然并非 Java 的语言成分,却是由 JDK 提供、跟 JVM 结合在一起的,实际上是 Java 虚拟机上的安全机制。可能这也是为什么 Hadoop 采用 Java 语言编写的原因之一,如果要自行设计和实现类似 JAAS 那样的机制和功能,那也并非易事。

JAAS 是“Java Authentication and Authorization Service”的缩写。其中 Authentication 说的是身份认证,主要涉及 Login 的过程,这可以是一般采用 Password 的简单 Login,也可以是采用 Kerberos 等比较复杂的第三方认证。而 Authorization,则主要是运行时的访问权限检验。JAAS 所做的检验不只局限于对于资源(例如文件)的访问权限,也包括对于行为,即程序代码的执行权限。按 JAAS 文档中的说法,传统上仅对资源进行的那种访问权限检验是“以用户为中心(user-centric)”的,现在则扩大到了“以代码为中心(code-centric)”。前者只考虑“这是谁,要对资源干什么”,而后者则扩大到了更为一般、更为广泛的“这是谁,要干什么”。其实,说“以代码为中心”还不如说“以行为为中心”或“以操作为中心”更明白一些。

权限检验需要有两方面的信息,一方面是“这是谁”,一方面是“谁可以做这个事”,或“允许这个用户做哪些事”。前者包含在一个称为 UGI,即 `UserGroupInformation` 的对象中。这个对象,这些信息,就好比用户的身份证件,应该是证不离身,永远跟着用户走;用户到了哪个节点上,它的 UGI 就应该跟到哪个节点上,就像随身携带身份证件一样。后者呢?对于资源,特别是文件,这信息在具体文件的目录项中,简单的如传统的 Unix 文件访问属性,说明了文件主、同组人、其他人对其分别有些什么访问权限,这是文件主可以设置的;复杂的如“访问控制名单(Access Control List, ACL)”,里面逐一列举了允许何人进行何种访问,具体文件的 ACL 也存储在文件系统中。而对于代码,即行为,则要有个“政策文件(Policy File)”,里面列举对什么人(或哪一组的人)授予什么权限。根据政策文件的内容,Hadoop 会生成相应的 `AccessControlList` 对象,其性质和文件系统中的 ACL 是一样的。其实,Linux 的 `/etc` 目录下有些文件的作用就相当于政策文件。

读者也许会问:那样岂不是很麻烦?是的,安全和方便、安全和效率,永远是矛盾的。所以 Hadoop 并不强制使用其安全机制,对于访问权限的检验,特别是对操作权限的检验是有条件的、可选择的。

最好还是结合实例来说明,我们先看看提交作业时的代码,对于 MapReduce 计算这是第一个关口:

```

public RunningJob submitJobInternal(final JobConf conf)
    throws FileNotFoundException, IOException {
    try {
        conf.setBooleanIfUnset("mapred.mapper.new-api", false);
        conf.setBooleanIfUnset("mapred.reducer.new-api", false);
        Job job = clientUgi.doAs(new PrivilegedExceptionAction<Job> () {
            //以 clientUgi 这个 UGI 的身份和名义执行作为参数的这个类所提供的 run()函数
            @Override
            public Job run() throws IOException, ClassNotFoundException,
                InterruptedException {
                Job job = Job.getInstance(conf);
                job.submit();
                return job;
            }
        });
        // update our Cluster instance with the one created by Job for submission
        // (we can't pass our Cluster instance to Job, since Job wraps the config
        // instance, and the two configs would then diverge)
        cluster = job.getCluster();
        return new NetworkedJob(job);
    } catch (InterruptedException ie) {
        throw new IOException("interrupted", ie);
    }
}

```

在这里,实际的作业提交,即 `job.submit()`,是放在 `doAs()` 函数中执行的。整个过程是这样:先动态(即临时)定义一个实现了 `PrivilegedExceptionAction` 界面的无名类,这个界面实际上只要求有个 `run()` 函数,所以这里就定义了一个 `run()` 函数。然后就创建一个此类对象,并以此为参数调用具体用户的 UGI 即 `UserGroupInformation` 对象中的 `doAs()` 函数。当然,这个 UGI 中记载着有关该用户身份的信息,就像我们的身份证或护照。而 UGI 的 `doAs()` 函数,则以具体用户的身份和名义调用这临时无名类对象的 `run()` 函数,使其提交作业。一旦程序从 `run()` 函数返回,就接着从 `doAs()` 返回,这个临时的无名类对象就由 JVM 回收了。

读者也许会问,从 `doAs()` 返回之后或者调用 `doAs()` 之前,程序是以谁的身份和名义在执行呢? 答案是这样:如果不是嵌套在另一个 `doAs()` 之中的话,这要看我们所在的 JVM 进程是由谁启动的。就 `job.submit()` 这个情景而言,这是在一个 App 中,那就要看是谁在宿主操作系统上启动了这个 App 作业,这又取决于用户是以谁的名义登录到宿主操作系统中。

不仅是作业提交,后面 AM 的创建、资源的分配、任务的指定和尝试、MapReduce 数据流的创建,每个重要的部位上都有这样的关卡。以 `YarnChild` 的启动为例,在其 `main()` 函数中,关键的一段代码是这样的:

```

try {

```



```

...
childUGI = UserGroupInformation.createRemoteUser(System
    .getenv(ApplicationConstants.Environment.USER.toString()));
// Add tokens to new user so that it may execute its task correctly.
//这里 createRemoteUser()的作用就是从下达的任务中抽取有关实际用户的信息
//并用这些信息创建一个 UserGroupInformation 对象
childUGI.addCredentials(credentials);

// set job classloader if configured before invoking the task
MRApps.setJobClassLoader(job);

logSyncer = TaskLog.createLogSyncer();

// Create a final reference to the task for the doAs block
final Task taskFinal = task;
childUGI.doAs(new PrivilegedExceptionAction<Object>() {
    @Override
    public Object run() throws Exception {
        // use job - specified working directory
        FileSystem.get(job).setWorkingDirectory(job.getWorkingDirectory());
        taskFinal.run(job, umbilical); // run the task
        //无论 Mapper 还是 Reducer,所有任务都是在 doAs()中运行的
        return null;
    }
});
} catch (FSError e) ...

```

我们知道, YarnChild 是在 AM 的安排下在某个普通节点上启动, 在一个独立的 Java 虚拟机上运行的。这个 JVM 进程一般是由系统启动的, 本来就具有特权用户所具有的各种权限。开始运行之后, 它要通过跟 AM 之间的“脐带(umbilical)”从作为母体的 AM 处领取任务。所领取的任务中就有关于用户即委托人的信息, 所以根据这些信息可以创建一个 UserGroupInformation 对象, 就是这里的 childUGI。这就好比我们去政府机关办事, 有些操作是要由公务员以他的身份和权限才能办的, 用我们自己的身份和权限办不成; 但是真正我们要办的事(如果可以办的话), 虽然也是由公务员经办, 但却是以我们自己的身份和权限办理的。随后的 childUGI.doAs(), 即 UserGroupInformation.doAs(), 就是这样。这个函数以 UGI 中所载用户的名义、身份和权限, (试图)去执行这个临时无名对象的 run() 函数。从代码中可见, 这个 run() 函数中所执行的操作主要是 taskFinal.run(), 就是执行 AM 交办的任务, 那就是 YarnChild 的全部价值所在。换言之, AM 交办的任务不是以系统的身份和权限, 而是以最终用户即委托人的身份和权限执行的。之所以每次调用 doAs() 时都是临时定义并创建一个对象, 是因为每次要做的事情不同, 当然在 run() 中调用的函数大多是静态定义的。

为什么说是“试图”去执行这个 `run()` 函数的代码呢？因为 `doAs()` 首先会检查所述的用户是否拥有进入这段代码的权限。有，事情可以办，才会以此用户的身份和权限执行这段代码，里面凡是遇有需要检验访问权限之处都使用这个 UGI 对象中所述用户和组别的权限。反之，如果没有，事情不可以办，那就根本不让进这段代码，直接就通过异常(Exception)机制退出了。

其实，在 `YarnChild.main()` 中，此前还另有一个 `doAs()`，其 `umbilical` 的建立就是在那个 `doAs()` 中完成的，那应该是以 AM 的名义，否则就不让进（设想如果使用者在键盘上打入了启动 `YarnChild` 的命令行）。

下面我们摘要看一下 `UserGroupInformation` 这个类及其 `doAs()` 方法的实现。

```
class UserGroupInformation {}
] AuthenticationMethod authenticationMethod
    //所采用的身份验证方法，可以是 SIMPLE、KERBEROS、TOKEN 等
] Configuration conf
] Groups groups    //这个用户属于哪一些组
] UserGroupInformation loginUser    //该用户登录时创建的 UGI
] String keytabPrincipal
] String keytabFile
] Subject subject    //这是对用户的身份描述
] User user    // User 是对 Principal 的扩充，用来保存用户的全名、缩略名等信息
] class UgiMetrics{}    //用来记录关于该用户的统计信息
] UserGroupInformation(Subject subject)    //构造方法
    > this.subject = subject
    > this.user = subject.getPrincipals(User.class).iterator().next()
    > this.isKeytab = !subject.getPrivateCredentials(KerberosKey.class).isEmpty()
    > this.isKrbTkt = !subject.getPrivateCredentials(KerberosTicket.class).isEmpty()
] ensureInitialized()
    > if (conf == null) {
    >+ initialize(new Configuration(), false)
    > }
] initialize(Configuration conf, boolean overrideNameRules)
    > authenticationMethod = SecurityUtil.getAuthenticationMethod(conf)
    >> String value = conf.get(HADOOP_SECURITY_AUTHENTICATION, "simple")
        //从配置中获取“hadoop.security.authentication”属性，默认为“simple”
    >> return Enum.valueOf(AuthenticationMethod.class, value.toUpperCase(Locale.ENGLISH))
        //将属性值转成大写（如“SIMPLE”），比对 AuthenticationMethod 类中的枚举值
    > HadoopKerberosName.setConfiguration(conf)
    > groups = Groups.getUserToGroupsMappingService(conf)
    > UserGroupInformation.conf = conf
] isSecurityEnabled()    //如果身份验证方法不是 SIMPLE，就表明安全机制是开启的
```

```

> return !isAuthenticationMethodEnabled(AuthenticationMethod.SIMPLE)
    //如果身份验证方法设定为 SIMPLE,就表示不启用安全机制
] doAs(PrivilegedAction<T> action)
    > logPrivilegedAction(subject, action) //在 Log 中加以记录
    > return Subject.doAs(subject, action) //转化为 Subject.doAs()
] doAs(PrivilegedExceptionAction<T> action)
    > logPrivilegedAction(subject, action) //在 Log 中加以记录
    > return Subject.doAs(subject, action) //转化为 Subject.doAs()
    > catch (PrivilegedActionException pae) {
    >+ ...
    > }

```

显然, `UserGroupInformation.doAs()` 其实就是 `Subject.doAs()`, 后者是由 JDK 提供的, `Subject` 这个类就是从 `javax.security.auth` 这个 package 中导入的, 用来描述行为主体, 即具体的用户。在这里, `UserGroupInformation.subject` 就是对于目标用户的身份描述, 目的就是要使当前线程从此以后就以目标用户的名义和身份运行。

注意, 在 `doAs()` 中并未涉及 `isSecurityEnabled()`, 那只是用于 Hadoop 这一层的安全机制, 而作为底层的 `Subject.doAs()` 中有 JAAS 自己的类似判定, 这是通过在启动 JVM 时在命令行加上选项“-Djava.security.manager”来启用的, 例如:

```
% java -Djava.security.manager -Djava.security.policy=someURL SomeApp
```

就是说, 启动 Java 虚拟机执行 `SomeApp`, 并开启安全机制, 启用 `SecurityManager`, 而使用的 Policy 文件则是 `someURL`。更详细的说明可参考 JAAS 的使用手册。

一般而言, 使用者知道这些, 照着 JAAS 的手册去做, 也就可以了。但是如果想要有更深入的了解, 那么 `Subject` 这个类的定义在 JDK 中。幸好有个开源的 `OpenJDK`, 使我们可以看一下它的摘要, 不过也不要陷得太深, 那已经超出了本书的范围。

```
[UserGroupInformation.doAs() > Subject.doAs()]
```

```

class Subject implements java.io.Serializable {} //来自 OpenJDK
] Set<Principal> principals
] Set<Object> pubCredentials
] Set<Object> privCredentials
] Subject()
    > this.principals = Collections.synchronizedSet(
        new SecureSet<Principal>(this, PRINCIPAL_SET))
    > this.pubCredentials = Collections.synchronizedSet(
        new SecureSet<Object>(this, PUB_CREDENTIAL_SET))
    > this.privCredentials = Collections.synchronizedSet(
        new SecureSet<Object>(this, PRIV_CREDENTIAL_SET))
] Subject(boolean readOnly, Set<?extends Principal> principals,

```

```

        Set<?> pubCredentials, Set<?> privCredentials)
    ] doAs(final Subject subject, final java.security.PrivilegedAction<T> action)
        > java.lang.SecurityManager sm = System.getSecurityManager() //如果没有开启就返回 null
        > if (sm!= null) sm.checkPermission(AuthPermissionHolder.DO_AS_PERMISSION)
            == SecurityManager.checkPermission(AuthPermissionHolder.DO_AS_PERMISSION)
        >> java.security.AccessController.checkPermission(perm)
        > AccessControlContext currentAcc = AccessController.getContext()
        > return AccessController.doPrivileged(action, createContext(subject, currentAcc))
    ] class AuthPermissionHolder{}

```

这里只摘要列出了一个 doAs(), 实际上有两个。这里所列的 doAs() 是以 PrivilegedAction 对象为参数的, 另一个则以 PrivilegedExceptionAction 对象为参数。如前所述, 这里的 PrivilegedExceptionAction 和 PrivilegedAction 其实都是只定义了一个 run() 函数的界面, 需要在调用 doAs() 时动态补上一个 run() 函数才可实体化成一个对象。二者的差别, 就在于一个带 Exception, 一个不带。这倒不是说一个可以发起异常而另一个不可以, 其实二者都有可能发生异常; 而只是一个会捕捉异常并加以一些处理, 另一个则不捕捉不处理, 将问题上交给调用者, 如果上一层也都不加捕捉和处理, 就会一直上传, 直到 Shell 的层面, 一般是使 JVM 的运行夭折。不过实际上当然不至于会那样。

从 Subject.doAs() 的代码摘要中可以看到: 如果开启了 JVM 的安全机制, 因而创建了一个 SecurityManager 对象 sm, 那就要调用 SecurityManager.checkPermission(), 从而调用 AccessController.checkPermission()。后者是这样的一个函数: 如果检验的结果是正面的, 允许执行给定的操作 action, 就顺利返回(无返回值); 如果不允许执行这个操作, 就会发生异常, 因而程序进入不了这个作为目标的 action 中。在这里, action 就是要通过 doAs() 实施的行为, 是通过 AccessController.doPrivileged() 完成的。但是当然, 由后者完成的行为是以具体 Subject 的身份实施的, 因为第二个调用参数是具体 subject 的 Context。

如果没有开启 JVM 的安全机制, 因而并未创建 SecurityManager 对象呢? 那就跳过了 checkPermission() 这一关, 允许进行所要求的操作。

AccessController, 顾名思义就是访问控制的实施者。而“访问控制”, 在这里实际上已被扩充成“活动控制”。SecurityManager 所提供的授权机制分成相辅相成互相配合的两个方面, 一方面是“授”, 另一方面是“控”, 这“控”的一面就是由 AccessController 实施的。

那么这里要查验是否允许的那个操作是什么呢? 并不是 action 本身, 而是 DO_AS_PERMISSION, 即“doAs”。也就是说, 要查验允不允许当前用户进行 doAs 这么一种活动, 即是否允许用别人的身份来执行某种操作。

下面是 OpenJDK 中 AccessController.checkPermission() 的摘要:

```

[UserGroupInformation.doAs() > Subject.doAs() > SecurityManager.checkPermission()
> AccessController.checkPermission()]

```

AccessController.checkPermission(Permission perm)

```

    //在我们这个情景中, 参数 perm 是 DO_AS_PERMISSION, 即“doAs”
    > if (perm == null) throw new NullPointerException("permission can't be null")

```

```

> AccessControlContext stack = getStackAccessControlContext()
//从堆栈上获取当前线程的 ACC
> if (stack == null) { //注意,这里的 stack 是个 AccessControlContext
>+ return //如果没有 AccessControlContext,那就默认允许,所以就直接返回了
> }
> AccessControlContext acc = stack.optimize()
// Take the stack-based context (this) and combine it with the
// privileged or inherited context, if need be.
> acc.checkPermission(perm) == AccessControlContext.checkPermission(perm)
// AccessController 的授权检验是由具体 AccessControlContext 实施的
>> if (perm == null) throw new NullPointerException("permission can't be null")
>> //iterate through the ProtectionDomains in the context. Stop at the first one
//that doesn't allow the requested permission (throwing an exception)
>> if (context == null) return
>> for (int i = 0; i < context.length; i++) {
>>+ if (context[i] != null && !context[i].implies(perm)) { //碰到一条不允许所述操作的规则
>>++ throw new AccessControlException("access denied " + perm, perm) //通过异常加以拒绝
>>+ }
>> }
>> return //如果所有规则都没有不允许所述操作,就正常返回,表示允许

```

这里要查验的是对于某项具体的操作是否允许放行。在我们这个情景中,参数 perm 是 DO_AS_PERMISSION,即“doAs”。可是,同样一个事,是否允许进行是因人而异的,这里针对的是谁呢?最初自然是当前用户,就是当前线程所属的用户。但若是在一个 doAs() 里面嵌套再碰上 doAs() 的时候就不一样了,因为此时这个线程可能已是以别人的身份在执行。

在 JAAS 这个层面上,有关具体用户授权的信息保存在一个 AccessControlContext 对象中。通常这个对象中的信息直接来自政策文件,但是正因为一个线程可能会 doAs() 别的用户,同一个用户的多个线程是可能会有不同 AccessControlContext 对象的。所以,这里的查验依据只能来自当前线程的 AccessControlContext 对象。进一步,本线程独有的信息只能要么放在堆栈上,要么放在 TLS (即互相独立的“线程本地存储”中)。这里通过 getStackAccessControlContext() 从堆栈上获取本线程的 AccessControlContext,显然是因为后者在本线程的堆栈上。

AccessControlContext 的核心是一个 ProtectionDomain 数组 context[], 数组的每个元素都是一个 ProtectionDomain, 后者的核心则是一个 PermissionCollection。

显然,PermissionCollection 意为各种 Permission 的集合。这是一个抽象类,具体加以扩充的则有 BasicPermissionCollection、FilePermissionCollection、SocketPermissionCollection、ExecPermissionCollection 等不同种类的 Permission 的集合。

Permission 又是个抽象类,直接或间接的具体化则有 AllPermission、FilePermission、ExecPermission、RuntimePermission、DynamicAccessPermission 等。

每一项 Permission 实质上都是一条规则,查验时依次实施这些规则,遇到第一条通不过

去的规则就发起 `AccessControlException` 异常,如果全部都通过就正常返回、通过了查验。可想而知,这些规则最终都来自具体的政策文件。

所以,前面以 `DO_AS_PERMISSION` 为参数调用 `AccessController.checkPermission()`,就是要查验当前线程此刻的有效 `AccessControlContext` 中是否含有关于“doAs”这种操作的授权。

回到前面 `Subject.doAs()` 的代码摘要,如果运行中通过了授权检验,下一步就是对 `AccessController.doPrivileged()` 的调用。注意这个函数的调用界面为:

```
doPrivileged(PrivilegedExceptionAction<T> action, AccessControlContext context)
```

而前面调用时的第二个实参则是“`createContext(subject, currentAcc)`”。这就是说,调用 `doPrivileged()` 时带下去的不是当前线程原有的 `AccessControlContext`,而是另外创建了一个 `AccessControlContext`,并且是专为目标用户 `subject` 而创建的。仍以 `YarnChild` 为例,为其启动 Java 虚拟机执行 `YarnChild.main()` 的可能是具体节点机上的特权用户,执行了 `doAs()` 之后,在节点机操作系统的眼中它的身份并未改变,仍是特权用户,但是在 JVM 的 `SecurityManager` 的眼中,以及在 Hadoop 的眼中,这个线程的访问权限已经降到了具体用户的水平,只能执行这个用户获得授权的操作、访问获得授权的资源。

我们不妨也看一下 `AccessController` 这个类的摘要,当然也是在 `OpenJDK` 中:

```
class AccessController {
    ] getContext() //获取本线程所属用户的 AccessControlContext
    ] checkPermission(Permission perm) //检查是否允许所要求的操作
    ] native doPrivileged(PrivilegedAction<T> action)
    ] native doPrivileged(PrivilegedAction<T> action, AccessControlContext context)
    ] native doPrivileged(PrivilegedExceptionAction<T> action)
    ] native doPrivileged(PrivilegedExceptionAction<T> action, AccessControlContext context)
    ] native getStackAccessControlContext()
    ] native getInheritedAccessControlContext()
```

这里有四个 `doPrivileged()`,两个用于 `PrivilegedExceptionAction`,另两个用于 `PrivilegedAction`。JDK 的 Java 语言源码中这四个函数都只有函数界面定义(申明)而没有具体实现,但是前面有个关键词 `native`。对于像这样前面带有 `native` 关键词的函数调用,都是通过 JNI 机制转入这个函数的 C 语言实现而完成的。这是因为下面的操作涉及 JVM 本身,靠 Java 语言已难以实现或无法实现,我们就不往下看了。

在前述的这个情景中,Hadoop 基于 `UserGroupInformation` 的安全措施 `doAs()` 是建立在 JAAS 的基础上,最终落实到 JAAS 基于 `AccessControlContext` 的同名操作,把 JAAS 这个机制用了起来。但是,要在 Hadoop 的集群环境中用好 JAAS,还得要补上一些针对集群环境特殊性的处理。

首先,集群内的节点间通信以及节点间的互信,就是个需要考虑的问题。在不同的具体条件下,这个问题可大可小。比方说,如果整个集群都集中在同一个机房中,由同一个单位使用,而且只运行 Hadoop,那么问题就比较小;但是如果可以同时为来自多个单位的用户服务,运行的程序也不只是 Hadoop,那么问题就可以很大。试想倘若节点间的通信都是明码,而

Mapper 节点的输出都是信用卡信息;或者某个节点正从另一个节点读取一个文件块,而文件块的内容都是病历信息,那要是被侦听的话后果就严重了。在分布计算中乃至互联网上,通信加密和节点间互信是普遍存在的问题,而且这二者常常是紧密联系在一起,所以早就有了一种集此二者于一体的机制和网络协议,称为 SASL(Simple Authentication and Security Layer),意为“简单身份验证与安全层”,具体的协议见 RFC 2222 和 RFC 4422。SASL 的思路是这样的:对于面向连接的应用层协议和传输层协议(例如 TCP)之间插入一层专门管身份验证和数据安全的框架,在这个框架中可以灵活插上具体的身份验证模块和数据加密模块。Hadoop 在这个环节上一方面提供了加密手段供选用,另一方面在节点之间也加上了身份验证(Authentication)的选项,我们在代码中看到当创建网络输出流和输入流时都可以加上一个 sasl 过滤层,就是采用了 SASL。

其次,在客户端/服务端即 C/S 架构中,访问权限的问题变得复杂起来。在 C/S 架构的系统中,服务端是应客户端的请求执行操作。虽然这个操作是由服务端在执行,但是实际的用户却是客户端上的那个用户。同样的道理,RPC 机制也使这问题变得复杂了。设想在两个 Java 虚拟机 Ja 和 Jb 之间,运行于 Ja 上的某个程序 Pa 通过 RPC 调用了运行于 Jb 上的另一个程序 Pb 所提供的某个函数 F,那么这个函数当然是在 Jb 上执行。再假定 F 在执行的过程中要访问某个文件,那么从操作系统的角度看,对此文件的访问权限取决于 Pb 的用户是谁、是由谁启动的。然而这却恰恰是错的,因为此时的函数 F 是由 Ja 上的另一个用户远程调用的,实际上是启动 Pa 运行的那个用户所启动。Hadoop 的操作大多都是跨节点的,更是跨 Java 虚拟机的,因而对于实际用户的身份描述应该跟着任务走,跟着 RPC 调用走,就好像我们在办什么事情时都要附上一张身份证复印件一样。比方说,具体的应用主管 AM 将用户提交的作业分解成许多任务后将其分发下达给具体的节点时,显然应该随同发送用户的 UserGroupInformation。再比方说,如果向一节点发送获取统计信息的请求,当然也应该说明究竟是谁想要了解这些信息。

17.2 UGI、Token 和 ACL

在 JAAS 的基础上,Hadoop 的安全机制主要集中在身份验证和权限管理两个方面,其中又以后者更为复杂和重要。这倒不是说身份验证就不重要,而是因为这方面的手段和措施,例如 Kerberos,基本上都是现成的,拿来用就是。另一方面,身份验证所解决的是准入问题,是允不允许进门的问题,位置相对集中和单一。而授权,所解决的是进了门以后的行为和访问两方面的权限控制问题,那范围就比较广了。

身份验证和授权在某种意义上是不可分割的,因为是否授权做什么事,或访问什么资源,就是根据两个因素确定的:第一是政策,例如针对具体文件的访问权限规定,文件主可以做什么,同组人可以做什么,其他人可以做什么,这就是政策(policy),或者说规定、规则。第二就是行为主体的身份,那已经在进门的时候验证过了,但是在控制具体的行为或访问资格时经常要查看这些信息。由于(经过验证的)身份信息经常要被查看,并且要做的事情常常是跨节点的,就得为经过身份验证的行为主体准备下一个 UGI,即 UserGroupInformation,这就好比是为其颁发了一本护照。对于行为控制,例如 doAs(),所依据的就是 UGI 中的信息和 policy 文件中的配置。在实施中,行为控制一般而言是比较宏观的;而对于资源的访问控制则比较精细,因为即使是同一行为主体,即同一用户,对不同资源的访问权限也不一样,而且不同资源的数

量也大。这样,要把所有资源的访问控制规则都集中在一个 policy 文件中是不现实的,所以从 Unix 时代以来都是把每个文件的访问权限规则都放在其目录项中,作为该文件的“元数据”的一部分。

但是这里有两个问题。

一个是在行为主体的身份信息这一边,对于许多行为控制和专门访问控制,尤其是访问控制,UGI 所提供的信息过于笼统、过于一般,还需要补充一些附加的专门信息才行。这就好比光凭一本护照是不够的,还需要有更为专门的、一事一授权的证件。这就是 Token 的作用。Token 这个词国内一般都译成“令牌”,其实这里未必有“令”的意思,但却确实有“牌牌”的意思(当然在特定条件下也可以用作“令牌”)。所以,在本书中我把 Token 称作“证章”。

另一个是在具体资源的访问权限规定这一边。传统的按“三种人”(文件主、同组人、其他人)确定访问权限的方法对于有些应用场景失诸过粗,实际上有必要针对具体用户列出其访问权限。于是就有了“访问控制名单(Access Control List, ACL)”的概念。可想而知,具体文件的 ACL 可长可短,要把它放在文件的目录项中就不合适的了,所以一般都是单独作为一个附加文件加以保存。

其实 Token 和 ACL 都不是新的概念,都是早已有之,在 Hadoop 中只是把它们用起来,以补 UGI 之不足而已。

为此,我们回顾并深入考察一下 UGI 中究竟有些什么信息。

```
class UserGroupInformation {}

//User and group information for Hadoop, wraps around a JAAS Subject.

[ AuthenticationMethod authenticationMethod //身份验证方法,如 kerberos
[ Configuration conf
[ Subject subject //代表着具体的用户
]] Set<Principal> principals //用户的 ID 集合
]] Set<Object> pubCredentials //公开的 Credential
]] Set<Object> privCredentials //私密的 Credential
[ User user //User 是实现了 Principal 界面的一个类,这是用户名
]] String fullName //长名
]] String shortName //短名
]] AuthenticationMethod authMethod //用户登录时所用的身份验证方法
]] LoginContext login //登录上下文
]] long lastLogin //最近一次登录的时间
[ Groups groups //用户属于哪些组
[ UserGroupInformation loginUser //用户登录时所产生的 UGI
```

Subject 对象代表着一个具体的用户,但是一个用户可能具有多个不同的 ID,即不同的 Principal。比方说有那么一个人,他可以用姓名(如“王和平”)登录,也可以用手机号登录,还可以用身份证号码或驾驶执照号码登录,甚至还可能用指纹或面容登录,所有这些不同的 principal 都可能在某种场合被用来代表同一个人。所以 Subject 的主要成分 principals 就是一个 Principal 的集合。不过要注意,Principal 在 openJDK 中定义为 interface,而不是 class,所以 set<principal>确切地说并非 Principal 对象的集合,而是“实现了 Principal 界面的某些

类的对象的集合”。那么有哪些类是实现了 Principal 界面的呢? 可以有很多, openJDK 中定义了 UnixPrincipal、NTDomainPrincipal、HttpPrincipal、LdapPrincipal 等, 应用软件中还可以自由定义别的类, 只要实现这个界面上规定的几个函数就行。下面的 User 就是 Hadoop 定义的一种 Principal。除 Principal 以外, 还有个 Credential 的概念, Credential 这个词本来就有“证物”的意思, JAAS 的文档中说用这个词来表示具体 Subject 的“security-related attributes and data”, 即有关安全的属性与数据, 比方说 Password, 比方说密钥, 再比方说指纹之类的生物特征。而密钥, 如果采用 RSA 等方法, 则又有公钥和私钥之分, 所以有些 Credential 是可以公开的, 但有些(大多)则是私密的, 所以这里分 pubCredentials 和 privCredentials 两种。注意, Credential 是多种多样的, 正如什么东西都可以成为证物, 所以它们的类型是 Object, 即任何对象, 而并未专门定义 Credential 这么一个类或界面。最后, Groups 是 Hadoop 定义的一个类, 意思当然就是用户所属的组的集合, 一个用户可以属于不止一个的组。

这样看来, UGI 所包含的信息似乎挺全面的了, 但是这些都只是身份信息, 里面并无有关授权的信息, 这意味着有关授权的信息一定集中在资源一侧。事实上在大部分情况下授权信息即访问控制信息也确实是集中在资源一侧, 例如文件系统对于“三种人”的访问权限规定, 就是存放在各个文件的目录项中; 如果觉得“三种人”的划分过于粗放, 想要规定得更精细, 那就用 ACL, 也是存放在文件系统中, 放在目标文件附近。即便是对于 doAs() 那样的行为控制, 也可以把 CPU 和相关的软件看作资源, 而载有权限控制信息的政策文件也存放在目标主机上, 也即资源一侧。

但是, 这样的授权毕竟还是比较粗放的, 有时候我们需要具有高度针对性的授权: 一事一授权, 而且有时间限制, 过了时间这个授权就失效了, 这有点像从前出门办事需要有单位开具的介绍信一样, 介绍信上要说明此人来办什么事、介绍信的有效期多长。这样, 资源一侧就不必每项资源都有个 ACL, 而且是完整的 ACL 了。前面我们把 UGI 比作护照, 上面只是身份信息; 而介绍信一类的证件、证书, 则进一步说明所针对的是什么具体的行为或资源。Token 就相当于这样的证件、证书, 就是这样的一块“牌牌”。

下面先介绍 Token, 然后再介绍一下 ACL。

Hadoop 的代码中定义了一个 Token 类, 下面是其数据部分的摘要:

```
//package org.apache.hadoop.security.token
class Token<T extends TokenIdentifier> implements Writable {}
] static Map<Text, Class<?extends TokenIdentifier>> tokenKindMap //TokenIdentifier 便查表
] byte[] identifier //具体 Token 内容的二进制映像
] byte[] password //用来证明发证人(而不是持证人)的身份
] Text kind //Token 的类型, 例如 HDFS_BLOCK_TOKEN、YARN_AM_RM_TOKEN
] Text service //服务方的名称, 例如服务端的 URI, 或[ IP 地址: 端口号 ]
] TokenRenewer renewer //用来办理 Token 延期
```

注意, 这是一个 Template, 即“模板”。所谓 Token 是关于具体 TokenIdentifier 的 Token, 有多少种不同的 TokenIdentifier, 就有多少种不同的 Token。所以, 实际上可以有很多种不同的 Token, 但是这些不同种类 Token 的结构在 Token 这一层都是一样的。

TokenIdentifier 是个抽象类, 其定义是这样:

```

abstract class TokenIdentifier implements Writable {}
] String trackingId
] abstract Text getKind()
] abstract UserGroupInformation getUser()
] getBytes() //获取本 TokenIdentifier 的二进制映像

```

至于具体的 `TokenIdentifier`, 则因具体的应用和资源种类而定。例如, HDFS 文件操作中用来授权访问具体数据块的 `Token`, 其 `TokenIdentifier` 为 `BlockTokenIdentifier`, 那就要增加好多针对数据块访问的信息:

```

class BlockTokenIdentifier extends TokenIdentifier {}
] Text KIND_NAME = new Text("HDFS_BLOCK_TOKEN")
] long expiryDate //Token 的截止期
] int keyId
] String userId //代表具体的用户, 据此可以创建 UGI, 见下面的 getUser()
] String blockPoolId //目标块所在的块池号
] long blockId //目标块的块号
] EnumSet<AccessMode> modes //授权的访问模式, 如 READ、WRITE
] byte[] cache
] getUser() //根据 TokenIdentifier 创建 UGI
> if (userId == null || "".equals(userId)) {
    //如果没有 userId, 或者 userId 为空, 就用 blockPoolId 加 blockId 作为 userId
>+ String user = blockPoolId + ":" + Long.toString(blockId)
>+ return UserGroupInformation.createRemoteUser(user) //创建 UGI
> }
> //在正常情况下, 就为 userId 所代表的用户创建 UGI
> return UserGroupInformation.createRemoteUser(userId) //创建 UGI

```

由此可见, 这种 `TokenIdentifier`, 以及由此而来的 `Token`, 就是专门用来授权访问具体数据块的。而且, 虽然这种 `TokenIdentifier` 中并不直接包含访问者的 UGI, 但是却有其 `userId`, 而 UGI 倒是可以根据 `userId` 创建出来的。

可想而知, 当 `NameNode` 告诉 `Client` 可以去哪些 `DataNode` 读取某个数据块的复份的时候, 应该给 `Client` 一个内含 `BlockTokenIdentifier` 的 `Token`, 让它持证前去读取。而如果另一个 `Client` 想要冒充前去诈读, 则会因为拿不出这样的 `Token` 而被拒绝。

再如 `AMRMTokenIdentifier`, 这显然是用于 AM 与 RM 之间的互动, 如请求分配资源:

```

class AMRMTokenIdentifier extends TokenIdentifier {}
//used by ApplicationMasters to authenticate to the ResourceManager.
] Text KIND_NAME = new Text("YARN_AM_RM_TOKEN")
] AMRMTokenIdentifierProto proto
] AMRMTokenIdentifier(ApplicationAttemptId appAttemptId, int masterKeyId) //构造方法
> builder = AMRMTokenIdentifierProto.newBuilder()

```

```

> if (appAttemptId!= null) {
>+ builder.setAppAttemptId(((ApplicationAttemptIdPBImpl)appAttemptId).getProto())
> }
> builder.setKeyId(masterKeyId)
> proto = builder.build() //构造 AMRMTOKENIdentifierProto 对象 proto
] getUser()
> if (proto.hasAppAttemptId()) {
>+ appAttemptId = new ApplicationAttemptIdPBImpl(proto.getAppAttemptId()).toString()
> }
> return UserGroupInformation.createRemoteUser(appAttemptId)
//若字符串 appAttemptId 为空会发生异常

```

从表面上看,这里似乎不像在 BlockTokenIdentifier 中那样有好几个关于目标的成分,而只有一个 proto 字段。但是实际上那是为 ProtoBuf 定义的一种报文格式,那就是 AMRMTOKENIdentifierProto,定义于 proto 文件 yarn_protos.proto 中(经 PB 编译之后会生成 Java 代码),那里面有两个成分:

```

message AMRMTOKENIdentifierProto {
    optional ApplicationAttemptIdProto appAttemptId = 1;    //第一个成分
    optional int32 keyId = 2 [default = -1];                //第二个成分
}

```

其中的第一个成分 ApplicationAttemptIdProto,则又可以分解:

```

message ApplicationAttemptIdProto {
    optional ApplicationIdProto application_id = 1;    //第一个成分
    optional int32 attemptId = 2;                    //第二个成分
}

```

所以,这里其实有 application_id、attemptId、keyId 三个成分,实际上构成了对于一次具体 AppAttempt 即 RMAppAttemptImpl 有关事务的授权。

再如关于 Container 的 ContainerTokenIdentifier 也是这样:

```

class ContainerTokenIdentifier extends TokenIdentifier {}
] Text KIND = new Text("ContainerToken")
] ContainerTokenIdentifierProto proto
] ContainerTokenIdentifier(ContainerId containerID, String hostName,
                             String appSubmitter, Resource r, ...)
> builder = ContainerTokenIdentifierProto.newBuilder()
> if (containerID!= null)
    builder.setContainerId(((ContainerIdPBImpl)containerID).getProto())
> builder.setNmHostAddr(hostName) //NodeManager 的 HostAddr
> builder.setAppSubmitter(appSubmitter) //作业提交者
> if (r!= null) builder.setResource(((ResourcePBImpl)r).getProto()) //具体的资源

```

```

> builder.setExpiryTimeStamp(expiryTimeStamp)
> builder.setMasterKeyId(masterKeyId)
> builder.setRmIdentifier(rmIdentifier) //ResourceManager 的 Identifier
> if (priority!= null) builder.setPriority(((PriorityPBImpl)priority).getProto())
> builder.setCreationTime(creationTime)
> proto = builder.build()
] getUser()
> if (proto.hasContainerId())
    containerId = new ContainerIdPBImpl(proto.getContainerId()).toString()
> return UserGroupInformation.createRemoteUser(containerId)

```

这里关于目标也只有一个字段 proto, 实际上是一个 ContainerTokenIdentifierProto 报文, 这个报文所包含的信息就更多了。

这样, 由具体 TokenIdentifier 所构成的 Token, 就带上了针对具体操作的授权信息, 行为者在实施相关操作时需要提交这样的 Token, 如果没有 Token 就提交身份信息。当然, 在服务端, 在实际发生具体操作的地方, 要根据行为者的 Token 判断其是否获得授权; 如果没有 Token 则可以考虑根据行为者的身份和具体资源有关授权的规定加以判断。除前述由 JAAS 的 SecurityManager 所提供的 checkPermission() 以外, 主要是由代表着各种资源的类所提供的 checkAccess() 方法加以实施。后面我们还要回到这个话题上来。

如上所述, 如果行为者不提交 Token, 那就要根据其身份和具体资源有关授权的规定加以判断。所谓具体资源有关授权的规定, 简单的就是三种人的三种访问权, 三三得九, 一共是 9 种不同的权限。如果要求更精细、更复杂的访问控制, 就得使用 ACL。一项资源的 ACL, 是一个访问控制名单, 或者说访问控制列表, 其中的每个表项 (即 AclEntry 对象), 都是一条具体的规定。下面是 AclEntry 类数据部分的摘要:

```

class AclEntry {}
] AclEntryType type // USER、GROUP、MASK、OTHER
] String name
] FsAction permission
] AclEntryScope scope // ACCESS、DEFAULT

```

简单地说, 就是授予名为 name 的用户 (或组) 以 permission 所述的访问权限。AclEntry 的类型分成 4 种, 其中的 USER、GROUP、OTHER 就是“三种人”, 表示 name 所代表的用户属于哪一种。MASK 类型则表示这个 AclEntry 被用作掩模, 凡是被其过滤掉的权限一律都被禁止。此外, 从另一个角度, AclEntry 所授予的权限又分两种 scope, 即 ACCESS 和 DEFAULT。前者是特别加以设置的; 后者则是从父 (目录) 节点继承而来, 如果本节点是目录节点就还可以往下传递。

所以, 给定一个 AclEntry, 我们可以知道: 这是从父节点继承下来的, 还是特别加以设置的; 这是针对个别 USER 即文件主的, 还是同一组的, 还是其他用户的, 或者只是一个掩模; 然后是用户名; 以及允许其进行的访问或操作, 即访问权限。

访问权限的类型是 FsAction, 这是个枚举类型:


```

enum FsAction {}
] NONE(" --- "), //0,三位都是 0
] EXECUTE("-- x") //1
] WRITE("- w - ") //2
] WRITE_EXECUTE("- wx") //3,最低两位为 1
] READ("r -- ") //4
] READ_EXECUTE("r - x") //5
] READ_WRITE("rw - ") //6
] ALL("rwx") //7,三位都是 1
] implies(FsAction that)
    > if (that != null) return (ordinal() & that.ordinal()) == that.ordinal()
    > return false

```

显然,这跟 Unix/Linux 中的文件访问权限格式是一致的,读、写、执三种权限用三个二进制位表示,一共是 8 种组合,相当于从 0 到 7 这 8 个数值。FsAction 提供一个方法 `implies()`,用来计算所给定的权限中是否包含行为者所要求的权限。比方说,FsAction 所给定的权限是 5,即 `READ_EXECUTE`,而行为者要求 1,即 `EXECUTE`,则 `implies()` 返回 `true`;但若要求 2,即 `WRITE` 则返回 `false`。

可见这里对于权限的划分与操作系统原有的划分并无不同,但是这样就可以更精细地为每个人进行个别的、量身定制的授权了。不过,要为每一项资源都设置 ACL,在 ACL 中为每个人都设置权限,那也是不现实的,所以 ACL 只是对于传统文件访问权限机制的补充,而并非替代。

HDFS 是个建构于宿主文件系统之上的更高层的文件系统,它的目录系统在 NameNode 上,对于宿主操作系统来说,HDFS 的目录,即 `FSImage`,只是一个(或几个)文件,宿主操作系统并不为 HDFS 的目录项单独提供 `Inode`。所以 HDFS 的 `Inode` 和目录等这整套机制都是 Hadoop 自行实现的,宿主操作系统的访问权限机制对 HDFS 的文件和目录起不到作用。可想而知,HDFS 也需要有自己针对“三种人”的那种访问控制机制,这是通过 `FsPermission` 实现的:

```

class FsPermission implements Writable {}
] FsAction useraction
] FsAction groupaction
] FsAction otheraction
] boolean stickyBit //黏滞标志位,控制着文件/目录的改名和删除,实质是对写入权的细化

```

显然这是在模仿 Unix/Linux 的文件访问权限设置。

而 ACL,则又是对 `FsPermission` 的补充。ACL 是作为 `Inode` 的一种属性实现的:

```

class AclFeature implements Inode.Feature {}
] ImmutableList<AclEntry> entries //一连串的 AclEntry
] AclFeature(ImmutableList<AclEntry> entries)
] getEntries()

```

显然, AclFeature 的数据部分就是一个 AclEntry 的 List,也就是一串 AclEntry。

一个文件的 FsPermission 是在创建文件时设定的,当然后面还可以变动,就像操作系统有 chmod() 系统调用一样。ACL 则一般都是后来补充的,客户端 DFSCClient 提供了一个方法 setAcl(), 使文件主可以设置 HDFS 文件的 ACL:

```
DFSCClient.setAcl(String src, List<AclEntry> aclSpec)
> checkOpen()
> namenode.setAcl(src, aclSpec) //通过 RPC 在 NameNode 上设置 HDFS 文件的 ACL
```

参数 src 是目标文件的路径名;aclSpec 则是一个 AclEntry 的 List,那就是 ACL。

当然,HDFS 文件的目录项在 NameNode 上,所以 DFSCClient 得要通过 RPC 将给定的 ACL 上传到 NameNode,在 NameNode 上设置目标文件的 ACL。RPC 的过程这里就不重复了,简而言之,是 NameNode 上的 RPC 机制会调用 FSNamesystem.setAcl():

```
[DFSCClient.setAcl() => FSNamesystem.setAcl()]
```

```
FSNamesystem.setAcl(final String srcArg, List<AclEntry> aclSpec)
> String src = srcArg
> nnConf.checkAclsConfigFlag()
> FSPermissionChecker pc = getPermissionChecker()
>> new FSPermissionChecker(fsOwnerShortUserName, supergroup, getRemoteUser())
> checkOperation(OperationCategory.WRITE) //在 HA 模式下,Standby 节点不允许写
> byte[][] pathComponents = FSDirectory.getPathComponentsForReservedPath(src)
> checkOperation(OperationCategory.WRITE)
> checkNameNodeSafeMode("Cannot set ACL on " + src) //SafeMode 下不允许设置 ACL
> src = resolvePath(src, pathComponents)
> checkOwner(pc, src) //是否为文件系统的创建者
>> checkPermission(pc, path, true, null, null, null, null) //后面还要介绍
> List<AclEntry> newAcl = dir.setAcl(src, aclSpec)
    == FSDirectory.setAcl(String src, List<AclEntry> aclSpec) //将 ACL 与目录项绑定
>> writeLock() //之所以称为 unprotectedSetAcl(),就是强调必须在加锁的条件下进行
>> unprotectedSetAcl(src, aclSpec)
>>> if (aclSpec.isEmpty()) {
>>>+ unprotectedRemoveAcl(src)
>>>+ return AclFeature.EMPTY_ENTRY_LIST
>>> }
>>> INodesInPath iip = getINodesInPath4Write(normalizePath(src), true)
>>> INode inode = resolveLastINode(src, iip)
>>> int snapshotId = iip.getLatestSnapshotId()
>>> List<AclEntry> existingAcl = AclStorage.readINodeLogicalAcl(inode) //读取原有 ACL
>>> List<AclEntry> newAcl = AclTransformation.replaceAclEntries(existingAcl, aclSpec)
//在原有 ACL 基础上
```

```
>>> AclStorage.updateINodeAcl(inode, newAcl, snapshotId)
>>> return newAcl
>>> writeUnlock() //解锁
> getEditLog().logSetAcl(src, newAcl) //记入 EditLog 日志
> resultingStat = getAuditFileInfo(src, false)
```

对于不同的资源,读、写、执这三项权限的含义,以及对于行为者身份的认定,可能会有所不同,检查的方法也可能有所不同,所以每一类资源原则上都应该有适合自身特性的许可检查,文件系统的许可检查就是 FSPermissionChecker。在设置 ACL 的过程中,就要创建一个 FSPermissionChecker,用来检查行为者是否就是目标文件的文件主,FSPermissionChecker 中包含了用户名、用户组以及行为者的远程 UGI 等信息。可想而知,设置 ACL 这个行为本身也是要受到访问控制的。

不难理解,ACL 是与(HDFS 所实现的)具体 INode 绑定的,但是也许这个 INode 原先就有 ACL,那样就要把原有的替换掉。最后,设置目标文件的 ACL 这个操作也要被记入 EditLog。

17.3 UGI 的来源和流转

在 Hadoop 的设计中,凡是 RPC 请求都要有 UGI,或者与之等价的信息随同发送,以示本次 RPC 调用是以何种身份发起,各种 RPC 请求报文的头部就包含了请求者的这些信息。但是身份的传递并不以一次 RPC 调用的返回为结束,凡是因此而引起的活动都有可能仍是以这个请求者的身份进行,这些活动又有可能发起别的 RPC 调用,从而形成身份的“扇出(fan-out)”。以作业提交为例,最初是某个节点上的具体应用通过 RPC 向 RM 节点提交作业, RM 节点接受了这个作业,本次 RPC 就返回了。但是用户的身份信息随同所提交的作业留了下来,此后 RM 节点在投运这个作业时的某些活动就会以这个用户的身份进行。然后 RM 指派在某个 NM 节点上创建这个作业的管理者 AM,这又是通过 RPC 实现的,这时候又要随同 RPC 请求发送作业主的 UGI(而不是 RM 本身的 UGI)。而 AM,则又要将这个作业所含的任务如 MapTask 分发到别的节点上。在这整个过程中要辗转发生很多次 RPC,每次都得携带着作业主即当初发动具体应用的那个用户的身份信息。也就是说,在这整个过程中流转的 UGI 信息应该是属于同一个用户的信息,这个 UGI 最初只能来自用户启动的具体应用。我们不妨以 examples 中的应用 WordCount 为例看一下用户 UGI 的来源:

```
WordCount.main()
> Configuration conf = new Configuration()
> Job job = new Job(conf, "word count") //创建 Job 对象
> job.setJarByClass(WordCount.class) //设置 Job 内容
> ...
> job.waitForCompletion(true) //提交作业,让 Job 在集群中运行
```

像别的应用一样,WordCount 也是先创建一个 Configuration,再创建一个 Job,然后对 Job 进行一些设置,例如设定 MapperClass、ReducerClass、FileInputFormat 等,最后就调用

job.waitForCompletion()。其中 Configuration 的信息主要来自各种配置文件,包括 core-default.xml、core-site.xml、hadoop-site.xml,也包括用于 Login 身份验证的 keytab 文件,这里面就可能有一些 credentials。对于所创建的 Job,这就是部分 credentials 的来源。但是还不止于此,更重要的是,具体 Job 的 Credentials 还来自从 JVM 获取的 UGI,而这里面的身份信息主要来自操作系统。我们看一下 Job 类的构造方法就可明白:

```
class Job extends JobContextImpl implements JobContext {}
] Job(Configuration conf, String jobName)    //构造方法
> this(conf) == Job(Configuration conf)
>> jc = new JobConf(conf)
>> this(jc) == Job(JobConf conf)
>>> super(conf, null) == JobContextImpl(Configuration conf, JobID jobId)
>>>> if (conf instanceof JobConf) this.conf = (JobConf)conf
>>>> else this.conf = new JobConf(conf)
>>>> this.jobId = jobId
>>>> this.credentials = this.conf.getCredentials()
>>>>> return credentials    //从 conf 中获取部分 Credentials,记入 Job.Credentials
>>>>> this.ugi = UserGroupInformation.getCurrentUser() //从 JVM 获取 UGI
>>>> creds = this.ugi.getCredentials()    //从 UGI 中获取 Credentials
>>>> this.credentials.mergeAll(creds) //将 UGI 中的 Credentials 并入 Job.Credentials
>>>> this.cluster = null
> setJobName(jobName)
```

Job 类是对 JobContextImpl 的扩充,所以凡是后者所有的成分前者都有,其中就包括 credentials 和 ugi。这里的 this.credentials 就是 Job.credentials,其内容是从两个来源,即 conf 和 ugi 合并而成的。而 ugi,则源自通过 UserGroupInformation.getCurrentUser()从 Java 虚拟机 JVM 获取的用户信息。为此我们可以看一下这个函数的摘要(注意 UserGroupInformation 是由 Hadoop 定义的一个类,JVM 本身不会创建这个类的对象):

```
UserGroupInformation.getCurrentUser()
> AccessControlContext context = AccessController.getContext()
    // AccessController 和 AccessControlContext 都是从 JDK 的 java.security 模块导入的
>> AccessControlContext acc = getStackAccessControlContext() //获取 AccessControlContext
    // getStackAccessControlContext()在 JDK 中,是个 native 方法,通过 JNI 转 C 函数
>>>> JVM_GetStackAccessControlContext(env, this)    //这是 JVM 代码中的一个 C 函数
>>> if (acc == null) return new AccessControlContext(null, true) //如果没有就创建一个
>>>> this.context = context    //null,这是一个无主的上下文
>>>> this.isPrivileged = isPrivileged    //true
>>>> return acc.optimize()
> subject = Subject.getSubject(context)
    //从 JVM 获取了 AccessControlContext,从中获取 Subject,即用户名
>>>> sm = System.getSecurityManager()
```

```

>> if (sm != null) sm.checkPermission(AuthPermissionHolder.GET_SUBJECT_PERMISSION)
>> if (acc == null) NullPointerException(
    ResourcesMgr.getString("invalid.null.AccessControlContext.provided"))
>> action = new java.security.PrivilegedAction<Subject>()
    ] run()
    > DomainCombiner dc = acc.getDomainCombiner()
    > if (!(dc instanceof SubjectDomainCombiner)) return null
    > SubjectDomainCombiner sdc = (SubjectDomainCombiner) dc
    > return sdc.getSubject()
>> return AccessController.doPrivileged(action) //执行上面的 run()
> if (subject == null || subject.getPrincipals(User.class).isEmpty()) {
    //如果从 JVM 拿不到用户身份信息,就要求用户登录
>+ return getLoginUser() == UserGroupInformation.getLoginUser()
>+> if (loginUser == null) {
>+>+ loginUserFromSubject(null) == UserGroupInformation.loginUserFromSubject(null)
>+> }
>+> return loginUser
> } else {
>+ return new UserGroupInformation(subject) //根据用户信息构建 UGI
> }

```

这里的斜体字部分都来自 openjdk-7。

这个函数调用 JDK 所提供的 AccessController.getContext() 和 Subject.getSubject(), 前者从 JVM 获取当前进程的访问控制上下文, 为此需要经由 JNI 机制调用 C 语言编写的函数, 通过 JVM_GetStackAccessControlContext() 从 JVM 的堆栈上获取这个上下文。后者则从所获得的上下文 context 中获取 Subject, 即操作系统所看到的用户信息。有了这些信息, 就可以构建出一个 UGI。

但是, 如果从获自 JVM 的访问控制上下文 context 中得不到用户信息, 那就得通过 getLoginUser() 获取应用层的用户登录信息。如果此前已经有过一个登录的过程, 那么该过程也会产生一个 UGI; 如果还没有经过这样的登录过程, 那就要求用户进行一次登录:

```
[UserGroupInformation.getCurrentUser() > loginUserFromSubject()]
```

```

UserGroupInformation.loginUserFromSubject(Subject subject)
> ensureInitialized() //确保 UGI 已经初始化
>> if (conf == null) initialize(new Configuration(), false) == UserGroupInformation.initialize(...)
> if (subject == null) subject = new Subject()
> LoginContext login = new LoginContext(authenticationMethod.getLoginAppName(),
    subject, new HadoopConfiguration())
> login.login() == LoginContext.login() //进行一次用户登录, 登录之后 Subject 就有内容了
> UserGroupInformation realUser = new UserGroupInformation(subject) //创建 UGI

```

```

> realUser.setLogin(login) //设置 UGI 中的 User.login 字段
> realUser.setAuthenticationMethod(authenticationMethod) //设置 UGI 中的 User.authMethod
> realUser = new UserGroupInformation(login.getSubject())
> String proxyUser = System.getenv(HADOOP_PROXY_USER)
//环境变量 HADOOP_PROXY_USER
> if (proxyUser == null) {
>+ proxyUser = System.getProperty(HADOOP_PROXY_USER)
> }
> loginUser = (proxyUser == null)?realUser : createProxyUser(proxyUser, realUser)
// loginUser 是 UGI 的一个内部成分,就是用户登录所形成的 UGI
> String fileLocation = System.getenv(HADOOP_TOKEN_FILE_LOCATION)
//环境变量“HADOOP_TOKEN_FILE_LOCATION”
> if (fileLocation != null) {
>+ Credentials cred = Credentials.readTokenStorageFile(new File(fileLocation), conf)
>+ loginUser.addCredentials(cred)
> }
> loginUser.spawnAutoRenewalThreadForUserCreds()

```

不过 login() 并不意味着必须是要求用户在键盘上登录,实际上也可以是从 keytab 文件读取用户名和 Password 的自动登录,当然还可以采用 Kerberos 身份认证。实际采用的身份认证方法和 LoginContext 都记录在 UGI 内的 User 对象中。

最后 UserGroupInformation.getCurrentUser() 所得到的是当前用户的 UGI,其内部成分 loginUser 是该用户登录时的 UGI。

这样,一个具体的作业就有了它的 UGI。

明白了 UserGroupInformation.getCurrentUser() 总是从 JVM 索取 AccessControlContext,我们又得再回顾一下 doAs()。我们在前面看到,UserGroupInformation.doAs() 是通过调用由 JDK 提供的 Subject.doAs() 实现其目标的,这已经不在 Hadoop 的代码中,但我们知道后者又调用 doPrivileged(PPrivilegedExceptionAction<T> action, AccessControlContext context),并为此通过 createContext() 创建了一个新的 AccessControlContext。所以,进入了 doAs() 之后,如果再有 UserGroupInformation.getCurrentUser(),则其返回的 UGI 来自最近一次进入 doAs() 的结果。

现在我们再回顾一下 RPC 机制的底层,并做定向的深化。我们知道(回到本书第 4 章),采用 ProtoBuf 时,RPC 的服务端是 ProtobufRpcEngine.Server,这个类的来历是这样的:

```

class ProtobufRpcEngine.Server extends RPC.Server {}
abstract class RPC.Server extends org.apache.hadoop.ipc.Server {}
abstract class Server {} //见~/org/apache/hadoop/ipc/Server.java
] CallQueueManager<Call> callQueue //RPC 调用队列
] Listener listener //Listener 线程
] Responder responder //Responder 线程
] Handler[] handlers //一组 Handler 线程

```


总之, `ProtobufRpcEngine.Server` 是对 `org.apache.hadoop.ipc.Server` 的扩充, 所以 `ProtobufRpcEngine.Server` 是个特殊的、具体的 `org.apache.hadoop.ipc.Server`, 后者所有的成分它都有。而后者内部就有个 `callQueue` 队列, 还有至少三个线程。其中 `Listener` 负责接受客户方的连接请求, 并建立连接; `Responder` 线程则对已建立的连接上到来的 RPC 请求做出初步的反应, 就是把请求报文接收进来, 挂在 `callQueue` 队列中; 而 `Handler` 线程则从 `callQueue` 队列中取出 RPC 请求加以处理。

我们先看一下 `Responder` 线程对于 RPC 请求的反应:

```
[Server.Responder.run() > doRunLoop() > doRead() > Server.Connection.readAndProcess() > processOneRpc()]
```

```
Server.Connection.processOneRpc(byte[] buf)
> processRpcRequest(header, dis)
>> rpcRequest = ReflectionUtils.newInstance(rpcRequestClass, conf)
>> rpcRequest.readFields(dis)
>> call = new Call(header.getCallId(), header.getRetryCount(),
                  rpcRequest, this, ProtoUtil.convert(header.getRpcKind()),
                  header.getClientId().toByteArray(), traceSpan)
>> callQueue.put(call)
```

`Responder` 线程往 `callQueue` 队列中挂, `Handler` 进程从这个队列中往外取:

```
Server.Handler.run()
> while (running) {
>+ Call call = callQueue.take() //只要队列中有,就往外取
>+ CurCall.set(call)           //这就是当前的 Call,即 CurCall
>+ if (call.connection.user == null) { //如果这个 Call 中没有用户信息,就直接调用 call()函数
>++ value = call(call.rpcKind, call.connection.protocolName, call.rpcRequest, call.timestamp)
>+ } else { //到来的 Call 中带有用户信息,就通过 doAs()以此用户的身份调用 call()函数
>++ value = call.connection.user.doAs(new PrivilegedExceptionAction<Writable>())
    ] run()
    > return call(call.rpcKind, call.connection.protocolName, call.rpcRequest, call.timestamp)
>+ }
> }
```

这里的 `call.connection.user` 是 `Call` 内部 `Connection` 的内部成分:

```
class Call implements Schedulable {}
] int callId;           // the client's call id
] Writable rpcRequest;   // Serialized Rpc request from client
] Connection connection // connection to client
]] SocketChannel channel
]] ByteBuffer data
```

```

]] UserGroupInformation user
]] UserGroupInformation attemptingUser //user name before auth
]] processConnectionContext(DataInputStream dis)
    > UserGroupInformation protocolUser = ProtoUtil.getUgi(connectionContext)
    > if (sasServer == null) {
    >+ user = protocolUser
    > } else {
    >+ user = UserGroupInformation.createProxyUser(protocolUser.getUserName(), realUser)
    > }

```

什么意思呢？RPC 调用请求来自与特定客户之间的连接 Connection，而这个连接在建立的时候就已经有了对方的 UGI，这是在 processConnectionContext() 中获取并保存的。而 Handler 线程在处理每个具体的 RPC 请求时，只要这个 Connection 是“实名制”的，call.connection.user 不是 null，后面的处理就都是在一个 doAs() 中进行。也就是说，RPC 请求一上岸，以后的活动就都是以客户的名义和身份进行的了。

以作业提交为例，RPC 请求在 RM 节点上经过 ProtobufRpcEngine 和 ProtoBuf 层的处理，到达应用层的第一关是 ClientRMService.submitApplication()，在这里：

```

ClientRMService.submitApplication(SubmitApplicationRequest request)
> submissionContext = request.getApplicationSubmissionContext()
> applicationId = submissionContext.getApplicationId()
> String user = UserGroupInformation.getCurrentUser().getShortUserName()
    == UserGroupInformation.getShortUserName()
> ...

```

这里 UserGroupInformation.getCurrentUser() 返回的是谁的 UGI？这个 getShortUserName() 得到的又是谁的用户名？考虑到在 RPC 的底层已经有过一次 doAs()，现在这是在第一次 doAs() 的内部，那这个用户就应该是作业的提交者。于是，前面客户端的 App 在创建 Job 时从 JVM 得到的 UGI，就跑到 RM 节点上来了。

作业提交是这样，别的 RPC 也是如此。这以后用户 UGI 随着创建 AM、资源本地化和 Task 投运的过程而从 RM 节点到其他节点的传播和扩散，就不言自明了。

17.4 Token 的使用

访问的授权和控制主要实现于资源这一侧，最基本的就是针对三种人的传统访问权限控制，更细一点的是 ACL。但是另外还有一种由行为者随身携带、一事一授权、类似于“介绍信”那样的机制，那就是 Token。而且 Token 的使用不仅可以针对资源，还可以针对行为。所以 Token 是一种重要的授权机制，尤其是在 Hadoop 这样的分布式系统中。

如前所述，所谓 Token 只是笼统的名称，真正起作用的是里面的内容即 TokenIdentifier。不同的 TokenIdentifier，就构成不同的 Token，这就好比不同的公司有不同的门禁卡。所以，Hadoop 的代码中把 TokenIdentifier 定义为抽象类，然后就在此基础上扩充成种种不同的

TokenIdentifier:

```
abstract class TokenIdentifier implements Writable {}
class BlockTokenIdentifier extends TokenIdentifier {} //用于 HDFS 数据块传输
class JobTokenIdentifier extends TokenIdentifier {} //用于 YarnChild 和 JobImpl 等
class AMRMTOKENIdentifier extends TokenIdentifier {} //用于 AM 与 RM 之间
class ContainerTokenIdentifier extends TokenIdentifier {} //用于 Container 投运
class NMTokenIdentifier extends TokenIdentifier {} //用于 NodeManager 上的某些活动
class ClientToAMTokenIdentifier extends TokenIdentifier {} //与 AM 的某些活动有关
class LocalizerTokenIdentifier extends TokenIdentifier {} //与资源本地化有关
```

不同的 TokenIdentifier, 携带的信息也不相同, 这就好比驾照上的信息跟医疗卡上的信息当然是不同的。

下面我们以 HDFS 中 DataXceiver 对象的 readBlock() 为例, 看一下 Token 在 readBlock() 函数中所起的作用。对于数据块的读取和传输, 这个“证章”就是基于 BlockTokenIdentifier 的 Token<BlockTokenIdentifier>。

```
DataXceiver.readBlock(ExtendedBlock block, Token<BlockTokenIdentifier> blockToken,
                      String clientName, long blockOffset, long length,
                      boolean sendChecksum, CachingStrategy cachingStrategy)

> previousOpClientName = clientName
> OutputStream baseStream = getOutputStream()
> DataOutputStream out = new DataOutputStream(
    new BufferedOutputStream(baseStream, HdfsConstants.SMALL_BUFFER_SIZE))
> checkAccess(out, true, block, blockToken, Op.READ_BLOCK,
              BlockTokenSecretManager.AccessMode.READ)
>> if (datanode.isBlockTokenEnabled) { //如果启用了 BlockToken 权限管理
>>+ datanode.blockPoolTokenSecretManager.checkAccess(t, null, blk, mode)
>>+> get(block.getBlockPoolId()).checkAccess(token, userId, block, mode)
    == BlockTokenSecretManager.checkAccess(token, userId, block, mode)
>> }
> // send the block
> DatanodeRegistration dnR = datanode.getDnRegistrationForBP(block.getBlockPoolId())
> updateCurrentThreadName("Sending block " + block)
> blockSender = new BlockSender(block, blockOffset, length, ...)
> writeSuccessWithChecksumInfo(blockSender, new DataOutputStream(getOutputStream()))
> long read = blockSender.sendBlock(out, baseStream, null); // send data
> ...
```

这里的第二个调用参数 blockToken 就是一个 Token<BlockTokenIdentifier>, 这个 Token 来自对方发来的 READ_BLOCK 请求。但是, 是否允许 sendBlock() 发送所请求的数据块则取决于程序能否通过 checkAccess() 这一关, 如果在 checkAccess() 中发生了异常, 程序

就到不了下面的 `sendBlock()`。所以这个 `checkAccess()` 就是对于所授权限的检查。从展开了的代码摘要看,如果没有启用 `BlockToken` 权限管理,那这个 `if` 语句就被跳过了,也就是默认通过了。但是如果启用了 `BlockToken`,那就取决于 `BlockPoolTokenSecretManager.checkAccess()`,进而取决于 `BlockTokenSecretManager.checkAccess()` 的查验结果:

```
[DataXceiver.readBlock() > BlockPoolTokenSecretManager.checkAccess()
>BlockTokenSecretManager.checkAccess()]

BlockTokenSecretManager.checkAccess(Token<BlockTokenIdentifier> token,
                                     String userId, ExtendedBlock block, AccessMode mode)
> id = new BlockTokenIdentifier()
> id.readFields(new DataInputStream(new ByteArrayInputStream(token.getIdentifier())))
> checkAccess(id, userId, block, mode)
>> if (userId != null && !userId.equals(id.getUserId())) //检查持证人与实际使用人是否相符
>>+ throw new InvalidToken(
    "Block token with " + id.toString() + " doesn't belong to user " + userId)
>> if (!id.getBlockPoolId().equals(block.getBlockPoolId())) //检查所属的 Namespace
>>+ throw new InvalidToken(
    "Block token with " + id.toString() + " doesn't apply to block " + block)
>> if (id.getBlockId() != block.getBlockId()) //检查数据块的 ID
>>+ throw new InvalidToken(
    "Block token with " + id.toString() + " doesn't apply to block " + block)
>> if (isExpired(id.getExpiryDate())) //检查有效期
>>+ throw new InvalidToken("Block token with " + id.toString() + " is expired.")
>> if (!id.getAccessModes().contains(mode)) //检查访问模式
>>+ throw new InvalidToken(
    "Block token with " + id.toString() + " doesn't have " + mode + " permission")
> if (!Arrays.equals(retrievePassword(id), token.getPassword())) { //检查 Token 中的口令
>+ throw new InvalidToken(
    "Block token with " + id.toString() + " doesn't have the correct token password")
> }
```

显然,就具体的数据块而言, `checkAccess()` 所查验的就是 `Token`, 而不是对于具体块文件的访问权限。这也很好理解,站在 `DataNode` 的立场,这个数据块(复份)是 `NameNode` 让我保存的,谁可以访问谁不可以访问,当然得听它的。

具体客户对于数据块的访问权限取决于它对这数据块所属文件的访问权限,而这些信息都记载在文件目录中,那是在 `NameNode` 上,所以最好的办法就是让 `NameNode` 给你一个 `Token`。用户要读文件时本来就得先去 `NameNode`, `NameNode` 会给它一个 `LocatedBlock`, 告诉它这个数据块的 ID、有几个复份、都存储在什么地方,同时就给它一个数据结构,那就是 `Token`。这个 `Token` 仅对读取这个特定数据块有效,而且过期失效,里面还带着发证人的口令。

不管是什么资源,只要是需有访问权限控制的,就应在它的类中定义一个 `checkAccess()` 方法。这当然也包括 `FSNamesystem`:

```
FSNamesystem.checkAccess(String src, FsAction mode)
> checkOperation(OperationCategory.READ) //检查是否允许读取有关文件目录的信息
>> if (haContext!= null) haContext.checkOperation(op)
> byte[][] pathComponents = FSDirectory.getPathComponentsForReservedPath(src)
>> return !isReservedName(src)?null : INode.getPathComponents(src)
> checkOperation(OperationCategory.READ) //经上述操作后再次检查是否允许读取信息
> src = FSDirectory.resolvePath(src, pathComponents, dir)
> if (dir.getINode(src) == null) FileNotFoundException("Path not found")
> if (isPermissionEnabled) { //如果没有启用就跳过以下的查验
>+ FSPermissionChecker pc = getPermissionChecker() //获取或创建 FSPermissionChecker
>+> UserGroupInformation user = getRemoteUser()
    == FSNamesystem.getRemoteUser()
>+>> NameNode.getRemoteUser()
>+>>> ugi = Server.getRemoteUser()
>+>>>> Call call = CurCall.get()
>+>>>> return (call!= null && call.connection!= null)?call.connection.user : null
>+> return new FSPermissionChecker(fsOwnerShortUserName, supergroup, user)
>+>> ugi = callerUgi
>+>> s = new HashSet<String>(Arrays.asList(ugi.getGroupNames()))
>+>> groups = Collections.unmodifiableSet(s)
>+>> user = ugi.getShortUserName()
>+>> isSuper = user.equals(fsOwner) || groups.contains(supergroup)
>+ checkPathAccess(pc, src, mode)
>+> checkPermission(pc, path, false, null, null, access, null)
>+>> checkPermission(pc, path, doCheckOwner, ancestorAccess,
    parentAccess, access, subAccess, false, true)
>+>>> if (!pc.isSuperUser()) {
>+>>>+ waitforLoadingFSImage()
>+>>>+ pc.checkPermission(path, dir, doCheckOwner, ancestorAccess, ...)
    == FSPermissionChecker.checkPermission(path, dir, ...) //见下述
>+>>> }
> }
```

Hadoop 为文件系统的访问权限查验定义了一个类,叫 `FSPermissionChecker`,把相关的操作都集中在这里。这个类的数据部分摘要是:

```
class FSPermissionChecker {}
] UserGroupInformation ugi
```

```
] String user
] Set<String> groups
] boolean isSuper
```

需要查验的时候,先把用户的信息收集在一个 `FSPermissionChecker` 对象中,然后就调用它的 `checkPermission()` 方法:

```
[FSNamesystem.checkAccess() > checkPathAccess() > checkPermission()
> FSPermissionChecker.checkPermission()]
```

```
FSPermissionChecker.checkPermission(String path, FSDirectory dir,
    boolean doCheckOwner, FsAction ancestorAccess,
    FsAction parentAccess, FsAction access, FsAction subAccess,
    boolean ignoreEmptyDir, boolean resolveLink)
> INodesInPath inodesInPath = dir.getINodesInPath(path, resolveLink)
> int snapshotId = inodesInPath.getPathSnapshotId()
> INode[] inodes = inodesInPath.getINodes()
> int ancestorIndex = inodes.length - 2
> for(; ancestorIndex >= 0 && inodes[ancestorIndex] == null; ancestorIndex--)
    //跳过可能存在的符号连接
> checkTraverse(inodes, ancestorIndex, snapshotId)
    //检查用户对目标文件路径中从根节点一直到其祖父节点的通行权
> INode last = inodes[inodes.length - 1]
> if (parentAccess != null && parentAccess.implies(FsAction.WRITE)
    && inodes.length > 1 && last != null) {
>+ checkStickyBit(inodes[inodes.length - 2], last, snapshotId)
    //如果 StickyBit 为 1,就只允许父节点(目录节点)的主人有写入权
> }
> if (ancestorAccess != null && inodes.length > 1) {
>+ check(inodes, ancestorIndex, snapshotId, ancestorAccess)
> }
> if (parentAccess != null && inodes.length > 1) {
>+ check(inodes, inodes.length - 2, snapshotId, parentAccess)
> }
> if (access != null) {
>+ check(last, snapshotId, access)
> }
> if (subAccess != null) {
>+ checkSubAccess(last, snapshotId, subAccess, ignoreEmptyDir)
> }
> if (doCheckOwner) {
```



```
>+ checkOwner(last, snapshotId)
> }
```

因为这是在 NameNode 上,是对整个目录在操作,所以对访问权限的查验不仅仅针对目标文件本身,而要查验整个文件路径,特别是对父节点(肯定是目录节点)的权限查验。

这样,就读取数据块复份操作而言,DataNode 认的是一事一授权的 Token。要求读取数据块的 Client,即 App,必须提交针对这个特定数据块的 Token。这个 Token 只能来自 NameNode,而 NameNode 对这个 Token 的授予则是基于这个 Client 对目标文件的访问权限。

第18章

Hadoop 的人机界面

供人们直接使用的系统须提供人机交互的手段,或称“人机界面(User Interface)”即 UI,更确切地说是“Man-Machine Interface”,使人们得以使用和管理这个系统或平台。比方说 Linux 上的 Shell,如 bash,就是个字符式的或者说“命令行”的人机界面;进一步则还有基于 X Server 的图形界面,例如我们常见的 Red Hat、Ubuntu 等 Linux 桌面发行版就是这样。现在也有很多系统的图形界面是建立在 Web 浏览器的基础上,即由系统提供一个 Web Server,使用者可以通过浏览器访问这个 Web Server,以此实现人与系统之间的交互,我们不妨称之为 Web 或浏览器式的人机界面。之所以可以这样,是因为作为客户端的浏览器通常都有图形功能,都是建立在图形基础上的,而浏览器与 Web Server 之间的通信和交互则已经标准化。这实质上是把系统本应在控制台上提供的图形功能嫁接到了远地的客户端上,因为那里的浏览器反正本来就有图形功能,这样可以使作为“服务器”的系统得以简化。其实命令行方式的人机界面也可以通过网络延伸到远地,例如从前的 telnet,后来的 rsh、ssh,以及像浏览器一样基于 HTTP 协议的工具 curl,就都是这样。

Hadoop 也不例外,它也提供了命令行式和浏览器式两种人机界面。

另外,系统在运行中会产生很多日志(Log)信息,这里说的是作为历史记录和告警信息的 Log,一般是写在磁盘上的 Log 文件中,虽然不是实时的屏幕显示,并非直接供用户观看阅读,但实际上终极的查看者也可以是人,所以 Log 通常都是用人们能直接阅读的文字形式输出的,也应看作是人机界面的一部分。

至于 MapReduce 的输出,那要看具体情况,但大多也是文字形式的。

18.1 Hadoop 的命令行界面

从功能和使用的角度看,Hadoop 分为两个子系统,为用户提供两个层次上的功能:一个是用来执行计算的 YARN 子系统,另一个就是提供文件服务的 HDFS 子系统。YARN 子系统主要是提供 MapReduce 计算,或者现在也可以提供简单的数据流式的 Stream 计算。HDFS 子系统则只是提供文件系统层面的服务,那是分布和容错的文件系统。这二者的设计都是专门针对 OLAP,即在线分析处理的,HDFS 中的文件基本上都是一次写入之后就转为只读,YARN 层与 HDFS 层少有往返交互(设想文件编辑),所以这二者之间的关系就比较松散,界线也就比较清晰,有点各走各的路那样的意味。

从管理的角度看,则主要是对于系统的控制、观察和统计,那也都是 YARN 归 YARN, HDFS 归 HDFS。

相比之下,操作系统,例如 Linux,就有个统一的命令行界面或者说字符式界面,那就是 Shell。一打开机器,我们就在 Shell 的界面上,你可以在这个统一的界面上输入命令 `ls`,也可以在这个界面上直接键入某个 App 的文件名以启动其运行。但是 Hadoop 并不提供这样一个统一的界面,而是充分利用宿主操作系统的 Shell,某种程度上以宿主操作系统的 Shell 为自己的 Shell,这样就显著简化了它的设计和实现。

要了解这两个不同的思路,我们可以想像两种不同的情景。第一种是这样:你在宿主操作系统的界面上键入“`java hadoop`”,Hadoop 一起来就以自己的 Shell 界面接管了宿主操作系统的 Shell 界面,此时你可以键入例如 `copyFromLocal`、`ls`,也可以键入例如 `pi`、`wordcount`,这些无非都是应用,对于 Hadoop 而言都是 Client,这就是一个统一的界面。这跟 Linux 的 Shell 是同样的风格,对于操作系统来说,这个进程一起来之后就一直在运行,一直处于一个“提示—接受—执行”的循环中,直到用户键入一条例如 `exit` 或 `bye` 那样的命令为止。第二种则是这样:你在宿主操作系统的 Shell 界面上键入例如“`java mapreduce wordcount`”这样的命令行以启动 YARN 子系统的 App,或者键入例如“`java dfs -ls`”以启动文件操作。在这里,对 YARN 的操作与对 HDFS 的操作是互不相关的,而实际的操作命令例如 `ls` 看上去就好像是参数或选项。对于操作系统来说,每次启动这样一个进程都只做一件具体的事,做完就退出了,用户又回到宿主操作系统的 Shell 界面,回到它的循环中。

这就是两种不同的思路,两种不同的设计。第二种的好处是简单易行,减少了软件开发的工作量。

Hadoop 系统中的各个大件,如 `ResourceManager`、`NodeManager`、`NameNode`、`DataNode`,都是在独立的 Java 虚拟机上运行,一台 Java 虚拟机就是宿主机操作系统上的一个进程。对于使用者来说,这些部件都带有 Server 的性质,使用者可以在同一宿主机上或者别的机器上运行一个带有 Client 性质的进程,通过进程间通信和 RPC 与这些部件交互来使用 Hadoop 平台。例如 MapReduce 作业,就是在宿主机操作系统的 Shell 界面上启动一个 Java 虚拟机来执行一个 App,这个 App 则通过 MapReduce 的 API 提交作业,实际上就起着 Client 的作用。所以,在 Hadoop 平台上 App 就是 Client。在这个过程中,Hadoop 利用宿主机操作系统的 Shell,使其实质上成为 YARN 子系统命令行界面的一部分。

HDFS 子系统的 Client 则是 `FsShell`,其人机界面同样也是在很大程度上利用了宿主机操作系统的 Shell。所以,以文件系统命令 `appendToFile` 为例,本来最合理、最符合人们使用习惯的命令行应该是“`appendToFile src1 src2 dst`”,但是在 Hadoop 所提供的人机界面上却是“`hdfs dfs -appendToFile src1 src2 dst`”,`appendToFile` 好像只是个选项。之所以是这样,就是为了尽量利用宿主机操作系统的 Shell。实际上 Hadoop 把概念上的 Shell 分成了内外两层,自己只提供内层,那就是 `FsShell`,而外层的 Shell 则直接利用宿主机操作系统的 Shell。这样就显著地简化了系统的设计和实现。注意,这里的 `hdfs` 是个脚本,`dfs` 其实就是 `FsShell`。

从平台管理的角度,则 Hadoop 还提供了 `DFSAdmin`、`DFSHAdmin`、`HSAdmin`、`RMAdminCLI`、`TraceAdmin` 等工具,都各有其命令行界面,这些工具实质上都是 App,也都是相应服务的 Client,对于宿主操作系统则都是独立的应用。

但是,通过命令行使用 Hadoop 其实是个挺麻烦的事,因为命令行既涉及具体子系统和模

块的许多选项和参数,又涉及启动 Java 虚拟机的许多选项和参数,每次都要从键盘打入实在是不胜其烦。所以 Hadoop 提供了几个脚本,供用户当作命令使用,其中最典型的应该算 hdfs。

我们在使用命令行例如“hdfs dfs -ls”时,形式上的命令是 hdfs,这就是个脚本。要了解这个脚本能做什么、有多复杂,从而有多灵活,最简单的办法是看一下它的 Usage,这相当于一个 Help 页面,下面是从脚本 hdfs 中剪下的一个片断:

```
function print_usage(){
    echo "Usage: hdfs [-- config confdir] COMMAND"
    echo "      where COMMAND is one of:"
    echo "    dfs                      run a filesystem command on the
                                file systems supported in Hadoop."
    echo "    namenode - format        format the DFS filesystem"
    echo "    secondarynamenode       run the DFS secondary namenode"
    echo "    namenode                 run the DFS namenode"
    echo "    journalnode              run the DFS journalnode"
    echo "    zkfc                     run the ZK Failover Controller daemon"
    echo "    datanode                 run a DFS datanode"
    echo "    dfsadmin                 run a DFS admin client"
    echo "    haadmin                  run a DFS HA admin client"
    echo "    fsck                     run a DFS filesystem checking utility"
    echo "    balancer                  run a cluster balancing utility"
    echo "    jmxget                    get JMX exported values from
                                NameNode or DataNode."
    echo "    mover                    run a utility to move block replicas across"
    echo "                                storage types"
    echo "    oiv                       apply the offline fsimage viewer to an fsimage"
    echo "    oiv_legacy                apply the offline fsimage viewer
                                to an legacy fsimage"
    echo "    oev                       apply the offline edits viewer
                                to an edits file"
    echo "    fetchdt                   fetch a delegation token from the NameNode"
    echo "    getconf                   get config values from configuration"
    echo "    groups                    get the groups which users belong to"
    echo "    snapshotDiff              diff two snapshots of a directory or diff the"
    echo "                                current directory contents with a snapshot"
    echo "    lsSnapshottableDir        list all snapshottable dirs owned by
                                the current user"
    echo "                                Use - help to see options"
    echo "    portmap                   run a portmap service"
    echo "    nfs3                       run an NFS version 3 gateway"
```

```

echo "  cacheadmin          configure the HDFS cache"
echo "  crypto              configure HDFS encryption zones"
echo "  storagepolicies      get all the existing block storage policies"
echo "  version               print the version"
echo ""
echo "Most commands print help when invoked w/o parameters."
}

```

这里把使用方法的提示定义成一个函数(过程),如果用户在键盘上光输入 hdfs,后面没有任何参数,则脚本中会调用这个函数 print_usage() 以显示使用提示:

```

if [ $# = 0 ]; then
    print_usage      # 如果 hdfs 命令行不带参数(包括具体操作命令本身),就显示 Usage
    exit
fi

```

可见,定义于 hdfs 的操作真是不少,这些就好像 Linux 上可以在 Shell 命令中启动的那些 Utility 工具软件。比方说这里的 fsck,那就相当于 Linux 的 fsck,只不过这里的 fsck 是针对 HDFS 文件系统的 fsck。注意这里又有“namenode -format”又有“namenode”,前者只用于 format,后者用于针对 namenode 的其他操作。另外我们也可从中看到,HDFS 文件系统中不但有 namenode、secondary namenode、datanode,还可以有 journalnode,那是用来记录文件操作流水账 EditLog 的。再如 balancer,就是启动 DataNode 节点间的(数据块复份)负载均衡。

这里所列举的 hdfs 操作命令,大多后面还要跟有针对具体命令的参数和选项。

用户键入命令行,启动 hdfs 脚本运行之后,这个脚本会根据具体的命令确定应该运行哪一个 Java 类。这看一下 hdfs 脚本中的这个片段就可明白:

```

if [ "$COMMAND" = "namenode" ]; then
    CLASS='org.apache.hadoop.hdfs.server.namenode.NameNode'
    HADOOP_OPTS="$HADOOP_OPTS $HADOOP_NAMENODE_OPTS"
elif [ "$COMMAND" = "zkfc" ]; then
    CLASS='org.apache.hadoop.hdfs.tools.DFSZKFailoverController'
    HADOOP_OPTS="$HADOOP_OPTS $HADOOP_ZKFC_OPTS"
elif [ "$COMMAND" = "secondarynamenode" ]; then
    CLASS='org.apache.hadoop.hdfs.server.namenode.SecondaryNameNode'
    HADOOP_OPTS="$HADOOP_OPTS $HADOOP_SECONDARYNAMENODE_OPTS"
elif [ "$COMMAND" = "datanode" ]; then
    CLASS='org.apache.hadoop.hdfs.server.datanode.DataNode'
    if [ "$starting_secure_dn" = "true" ]; then
        HADOOP_OPTS="$HADOOP_OPTS -jvm server $HADOOP_DATANODE_OPTS"
    else
        HADOOP_OPTS="$HADOOP_OPTS -server $HADOOP_DATANODE_OPTS"
    fi
fi

```

```

elif ...
...
elif [ "$COMMAND" = "dfs" ] ; then
    CLASS = org.apache.hadoop.fs.FsShell
    HADOOP_OPTS = "$HADOOP_OPTS $HADOOP_CLIENT_OPTS"
elif ...
...
else
    CLASS = "$COMMAND"
fi

```

如果 `hdfs` 命令行中的 `COMMAND` 是 `namenode`, 那么相应有待执行的 Java 类, 即变量 `CLASS` 的值, 就是 `NameNode`, 此时的命令行选项就是 `HADOOP_OPTS` 和 `HADOOP_NAMENODE_OPTS` 这两个变量的值; 而如果命令 `COMMAND` 是 `zkfc`, 那就是 `DFSZKFailoverController`, 此时的命令行选项则是 `HADOOP_OPTS` 和 `HADOOP_ZKFC_OPTS` 这两个变量的值。余可类推。注意, 在这个条件语句链末尾的 `else` 分支是“`CLASS=$COMMAND`”, 这说明, 如果命令行中的 `COMMAND` 与脚本 `hdfs` 中所提供的操作全都对不上, 那就是自定义操作, 用户应提供相应的 Java 类。

这样, 确定了与命令相对应的 Java 类 `CLASS` 以后, 如果采用常规的 Java 虚拟机即 `java` 加以执行, 最后就是:

```
exec "$JAVA" -Dproc_$COMMAND $JAVA_HEAP_MAX $HADOOP_OPTS $CLASS "$@"
```

这里变量 `JAVA` 的值就是 `java`; 选项 `-Dproc_$COMMAND` 表示进程的名称, 例如“`proc_fsck`”; 变量 `JAVA_HEAP_MAX` 的值表示内存 `Heap` 的大小, 是命令行的参数之一; 变量 `HADOOP_OPTS` 的值定义于另一个脚本 `hadoop-config.sh` 中, 其实有好长一串, 那都是命令行可选项。而变量 `CLASS` 的值, 那就是上面确定的 Java 类名称和路径。至于最后的“`$@`”, 则表示如果命令行中还有别的参数和可选项就都照抄在后面。

这还只是摘要剪辑, 实际上还更复杂。对于 `HDFS` 如此, 对 `YARN` 也是类似。这些脚本也应视作 Hadoop 源码的一部分, 其实是 Hadoop 命令行界面的一部分。

上面所述都是控制台模式的命令行人机界面, 这些命令行人机界面都是在宿主机上实现的, 都要用到宿主机的键盘和显示器。可是 Hadoop 集群的机器一般都不是普通的台式机, 而是装在机架上的服务器, 不会都带键盘和显示器, 怎么办呢? 最简单而现成的办法是在用户所在(集群之外)的机器上使用 `ssh`, 即“安全的远程 Shell(第一个 `s` 表示 `secure`)”。使用 `ssh` 可以通过网络远程连接到集群中的任何一台机器上。Unix 上很早就有 `telnet`、`rlogin`、`rsh` 等远程工具, 可以通过网络远程登录到别的 Unix 机器上, 把本机的终端用作目标机的终端, 仿佛就是把目标机的 Shell 拉到了本地, 用起来很是方便。然而这些工具的安全性都不够好, 工作时的传输内容没有加密, 所以后来又有了 `ssh`。

通过 `ssh` 连接到目标机上, 其实与 Hadoop 并无直接的关系, 那只是连接到了目标机的操作系统, 并且需要对方有个服务进程 `sshd` 在运行才行。一旦连接到了目标机的操作系统, 本地的键盘和显示器(窗口)就好像连到了对方的机器上, 可以在对方机器上启动 Java 虚拟机以

运行例如 MapReduce 作业或 HDFS 文件系统管理了。

18.2 Hadoop 的 Web 界面

如果安于使用命令行,用户实际上可以通过 ssh 连接到集群中的任何一台机器,然后在对方的 Shell 上启动运行种种 Hadoop 命令行。但是这在互联网技术如此发达的今天毕竟是有不合时宜的了。所以 Hadoop 也提供了基于 HTTP 和 HTTPS 协议的 Web 界面,使用户可以通过浏览器与 Hadoop 平台交互。

为此目的,集群中每个节点上的每个作为独立进程(Java 虚拟机)提供服务的类,例如 NodeManager、NameNode 等等,原则上都得要有个专门提供 Web 服务的对象(模块)即 Server,或者说“服务器”。Server 这个词有时候是指物质意义上的“器”,那就是一台用来提供服务的机器;有时候则只是指提供服务的软件模块(而并非物质意义上的“器”)。这里所说的 Server 是软件模块,通常是一个 Java 类。

所谓 Web 服务,就是接收用户发来的 Web 请求(报文),然后根据请求产生并发回相应的响应(报文),那一般就是一个“网页(Web Page)”。最简单的网页就是一个静态存在的文件,那就是静态网页,这跟远程 cat 一个字符文件没有多大区别,只是文件内容须符合 HTTP 的格式规定。复杂一点的,则在静态的网页中嵌有一些动态生成的内容,例如一些统计数据或状态信息。再复杂一些,就是以动态生成的内容为主了,那就相当于远程启动执行一个应用软件并发回其输出。就 Hadoop 而言,提供 Web 服务的目的是要为用户提供远程的使用界面,起到类似于 Shell 的作用,所以必然是以动态生成响应内容为主。另外,既然 Web 服务起着类似于 Shell 的作用,它就应该是一个“有源”的进程或线程,而不只是一组“无源”的库函数。

另外,现下的 Web 服务基本上都采用 HTTP 协议和 HTML 语言,所以 HTTP 服务与 Web 服务实际上是同义词。但是将来在互联网上也许还会使用不同的规程和语言,所以严格说来 HTTP 服务只是 Web 服务的子集和特例,后者的范围可以更广。

早期的 Web 服务都采用 CGI(Common Gateway Interface)技术,Web 服务器在接收到 HTTP 请求之后就创建一个进程,根据请求中的 URL 执行相应的可执行程序,例如 cat,然后把程序的输出转换包装成 HTML 语言的响应报文并发回。在这种技术中,响应信息的生成是需要创建一个进程来完成的,这当然很灵活也很安全(因为每个进程的内存空间都互相独立),但是每次服务都要另行创建一个进程,并且在执行过程中可能会有很多次的进程调度和切换(一个服务器可能同时有几千、几万个并发的服务正在进行中),系统的开销未免太大,效率太低。对这种技术的改进是以线程取代进程。线程是“轻量级”的进程,同一进程内部的所有线程都共享同一内存空间,其创建和调度切换的系统开销都要小得多,这当然有利于提高运行效率。再进一步,那就是连线程也不是临时创建的,而是由线程池中通用的线程根据具体的请求加载相应的程序模块即对象并加以调用。为此目的,这些有待加载的对象都应该有基本相同的调用界面和属性,都应该是对于同一抽象类的扩充和实现。这样的对象,就是所谓 Servlet 对象,其作用类似于动态加载的程序库 DLL。不过,不同的 DLL 不必有相同的调用界面,只要各自预先定义好界面就可以了,而 Servlet 则必须实现相同的界面。可见 Servlet 其实是个界面,所谓“Servlet 对象”其实是“实现了 Servlet 界面的某类对象”。至于调用此类 Servlet 以动态生成响应信息的模块,则称为“Servlet 引擎”,那显然是个线程。

在实际的操作中,一个(动态产生响应信息的)任务常常不是只由一个 Servlet 完成,而是由一串 Servlet 接力完成的,这有点像 Linux 的“管道(Pipe)”流水线,流水线中的每一个节点称为一个“过滤器(Filter)”,其核心就是 Servlet。这样,就可以用一组 Servlet 像积木块一样灵活搭建出功能强大的加工链。

由于 Servlet 及其引擎在 Web 服务器中的重要性和普遍性,Java SDK 之类的 package 中提供了 Servlet 界面和一些基础性的类和机制的实现,所以编程时凡涉及 Web 服务和 Servlet 都可以导入和利用相关的 package。在 Hadoop 的代码中,有关 Servlet 的定义是从 JDK 即 javax 中导入的。相关的代码则部分来自 com.google.inject,那显然是由 Google 提供的;而有关 Web 服务器的代码则导入 org.mortbay.jetty,那是 Eclipse Foundation 旗下的一个开源项目,与广泛采用的开源 Web 服务器 Tomcat 类似。

可想而知,Hadoop 中那些涉及 Shell 人机界面的部件和子系统,至少 ResourceManager、NodeManager、NameNode、DataNode,都应该有自己的 Web 服务器。更具体地说,是这些进程中都应该有自己的 Web 服务线程。当然,由于 YARN 和 HDFS 的业务不同、主节点与从节点的角色和作用不同,它们的 Servlet 也会不同,这又可能影响到它们的 Servlet 引擎即服务器也有所不同。

我们以 ResourceManager 的 Web 界面为例考察其创建和运行,搞懂了这个以后读者就可举一反三,其他几个部件和子系统的 Web 界面也就容易理解了。ResourceManager 内部有个成分 webApp,是个 WebApp 类的对象。但是 WebApp 是个抽象类,所以实际上这个 webApp 是个扩充了 WebApp 的 RMWebApp 对象。顾名思义,这是 RM 节点上的一种 Web 应用,对于远程的用户而言则是一种服务。而抽象类 WebApp,则又是对 ServletModule 类(从 com.google.inject 导入)的扩充。所以,实际上 RMWebApp 是个基于 Servlet 的模块。下面是从 Web 服务角度对 ResourceManager 类的摘要:

```
class ResourceManager{
] WebApp webApp //在 RM 中实际上是 RMWebApp
] serviceStart()
    > if (this.rmContext.isHAEnabled()) transitionToStandby(true)
    > else transitionToActive()
    > startWepApp() //创建 RMWebApp 对象
    > if (getConfig().getBoolean(YarnConfiguration.IS_MINI_YARN_CLUSTER, false)){
    >+ int port = webApp.port()
    >+ WebAppUtils.setRMWebAppPort(conf, port)
    > }
    > super.serviceStart()
```

显然,这里面包含着一个 WebApp 类的对象 webApp,并且在初始化之后的 serviceStart() 中通过 startWepApp() 创建了这个 WebApp 对象,实际上是 RMWebApp 对象:

```
abstract class WebApp extends ServletModule {}
class RMWebApp extends WebApp implements YarnWebParams {}
```

之所以在这里是 RMWebApp,当然是因为这是在 RM 内部。可想而知,在不同的部件和

子系统中会有对于 WebApp 的不同扩充。要启动 RM 的 Web 服务,首先要创建一个 RMWebApp 对象,所以这里 startWepApp() 的作用主要在于创建 RMWebApp 对象 webApp。

```
[ResourceManager.main() > ResourceManager()
> ResourceManager.serviceStart() > startWepApp()]

ResourceManager.startWepApp()
> Configuration conf = getConfig()
> boolean useYarnAuthenticationFilter = conf.getBoolean(
    YarnConfiguration.RM_WEBAPP_DELEGATION_TOKEN_AUTH_FILTER,
    YarnConfiguration.DEFAULT_RM_WEBAPP_DELEGATION_TOKEN_AUTH_FILTER)
    // 从系统配置中获取关于是否要使用 Kerberos 身份认证的设置
> String authPrefix = "hadoop.http.authentication."
> String authTypeKey = authPrefix + "type"
> String filterInitializerConfKey = "hadoop.http.filter.initializers" //配置项的名称
> String actualInitializers = ""
> Class<?>[] initializersClasses = conf.getClasses(filterInitializerConfKey)
    // 从系统配置中获取关于身份认证所用 Filter 的设置(可以有多个)
    // 在配置文件 core-default.xml 中设置为 StaticUserWebFilter
> if (initializersClasses != null) {
>+ for (Class<?> initializer : initializersClasses) {
>++ ... //调用这些 Filter 的基本初始化程序
>+ }
>+ if (UserGroupInformation.isSecurityEnabled() && useYarnAuthenticationFilter
    && hasHadoopAuthFilterInitializer && conf.get(authTypeKey, "")
    .equals(KerberosAuthenticationHandler.TYPE)) {
>++ ... //如果启用安全机制,则须进行更多的初始化操作,此处从略
    //有兴趣或需要的读者可自行阅读相关代码
>+ }
> } //end if (initializersClasses != null)
> String initializers = conf.get(filterInitializerConfKey)
> if (!UserGroupInformation.isSecurityEnabled()) { //如未启用安全机制,就修改配置
>+ if (initializersClasses == null || initializersClasses.length == 0) {
    //如未设定 initializersClasses,就只有 RMAuthenticationFilterInitializer 一个
>++ conf.set(filterInitializerConfKey, RMAuthenticationFilterInitializer.class.getName())
>++ conf.set(authTypeKey, "simple") //设置认证模式为 simple
>+ } else if (initializers.equals(StaticUserWebFilter.class.getName())) {
    //如果设定了 initializersClasses,就是 RMAuthenticationFilterInitializer + initializers
>++ conf.set(filterInitializerConfKey,
    RMAuthenticationFilterInitializer.class.getName() + "," + initializers)
```

```

> ++ conf.set(authTypeKey, "simple") //设置认证模式为 simple
> + }
> } //end if
> //准备创建 RMWebApp 对象,先创建一个 WebApps.Builder 对象 builder
> Builder<ApplicationMasterService> builder =
    WebApps.$for("cluster", ApplicationMasterService.class, masterService, "ws")
        .with(conf)
        .withHttpSpnegoPrincipalKey(
            YarnConfiguration.RM_WEBAPP_SPNEGO_USER_NAME_KEY)
        .withHttpSpnegoKeytabKey(
            YarnConfiguration.RM_WEBAPP_SPNEGO_KEYTAB_FILE_KEY)
        .at(webAppAddress) // WebApps.$for() 返回一个 Builder 对象,
                            // 随后的操作都只是对该对象进行设置,每次都返回该对象本身
> String proxyHostAndPort = WebAppUtils.getProxyHostAndPort(conf)
                            // 获取代理服务器的网址和端口号
>> String addr = conf.get(YarnConfiguration.PROXY_ADDRESS) // "yarn.web-proxy.address"
>> if(addr == null || addr.isEmpty()) {
>> + addr = getResolvedRMWebAppURLWithoutScheme(conf)
        // 在配置块中查找网址 "yarn.resourcemanager.webapp.address", 默认为 "0.0.0.0 : 8088",
        // 若用 HTTPS 则为 "yarn.resourcemanager.webapp.https.address", 默认为 "0.0.0.0 : 8090"
>> }
>> return addr // 返回网址字符串
> if(WebAppUtils.getResolvedRMWebAppURLWithoutScheme(conf)
    .equals(proxyHostAndPort)) { // 如果两个 URL 相符
    // 此种情况下的“代理服务器”就依附在 RM 内部,成为 RMWebApp 的一部分
    // 所以要安排把 WebAppProxyServlet 添加到 RMWebApp 中
    // 否则 WebAppProxyServlet 应在一个独立的代理服务器 WebAppProxyServer 中
> + if (HAUtil.isHAEnabled(conf)) fetcher = new AppReportFetcher(conf)
> + else fetcher = new AppReportFetcher(conf, getClientRMService())
    // fetcher 是一个 AppReportFetcher 对象,用来获取具体应用的 ApplicationReport,
    // 其创建方式因 HA 是否启用而有所不同
> + builder.withServlet( // 指定添加一个 Servlet,即 WebAppProxyServlet:
    ProxyUriUtils.PROXY_SERVLET_NAME, // 这就是“proxy”
    ProxyUriUtils.PROXY_PATH_SPEC, WebAppProxyServlet.class)
    // 将 WebAppProxyServlet 及其路径“/proxy/*”添加到集合 servlets 中
    // 这个集合中的所有 servlet 都会被加入 HttpServer2 中
> + builder.withAttribute(WebAppProxy.FETCHER_ATTRIBUTE, fetcher)
    // 与 Servlet 配套的 fetcher 是 AppReportFetcher
> + String[] proxyParts = proxyHostAndPort.split(":")
> + builder.withAttribute(WebAppProxy.PROXY_HOST_ATTRIBUTE, proxyParts[0])

```

```
> } //end if
> webapp = new RMWebApp(this) //webApp 实际上是个 RMWebApp
>> this.rm = rm //RMWebApp 的构造方法很简单
> webApp = builder.start(webapp) == WebApps.Builder.start(WebApp webapp)
//启动之后就成为 ResourceManager.webApp,见后述
//这样,webApp 就与 builder 所构建的 Servlet 链绑定了
```

这段代码的前半部有关 Kerberos 身份认证。如前所述,Web 服务常常不是由单个 Servlet 完成,而是由一个 Filter 链完成的;Filter 链中的每个节点都是 Filter,其核心就是个 Servlet。这样,就可以根据需要灵活搭建合适的 Filter 链。这里是要在 Filter 链中插入用于身份认证的 Filter,而具体是什么 Filter 则要看配置块中对于 filterInitializerConfKey 的设置,并须加以初始化。另外,这里还根据具体的情况对配置项做出一些修正,为后面创建 HttpServer2 对象做好准备。实际上这里插入的是 StaticUserWebFilter 内部的 StaticUserFilter。前者是个 FilterInitializer,后者即 StaticUserFilter 才是个 Filter,有兴趣或需要的读者可自行阅读相关的代码。同样,RMAuthenticationFilterInitializer 也只是为了把 RMAuthenticationFilter 添加到 Filter 链中。

代码的后半部旨在创建 RM 的 WebApp。注意这里的 WebApp 和 WebApps 是两码事,WebApps 是用来帮助创建各种 WebApp 对象的。不过 WebApp 是个抽象类,这里实际创建的是经过扩充落实的 RMWebApp 类对象。程序中首先通过 WebApps.\$for() 创建一个 Builder 对象,这是用于创建具体 WebApp 的工具。注意,这个 \$for() 是 WebApps 类所提供的的一个方法,用来创建(适用于 ApplicationMasterService 的)Builder 对象。随后的 with()、at()、withServlet() 等方法则只是用来设置某些属性,相当于对工具进行的加工准备。这些方法均返回这个 Builder 自身,所以可以把这些操作都串起来。其中的 Servlet 当然是核心所在,这里采用的是 WebAppProxyServlet。与之配合的是 AppReportFetcher,那是用来从 RM 获取 ApplicationReport 的。

WebAppProxyServlet,以及与之配套的 AppReportFetcher,显然与 AppReport 有关,其作用是让用户可以通过 Web 界面获取关于 App 的统计信息。这里的 if 语句并非关于要不要有 WebAppProxyServlet,而是关于是否把这个 servlet 注入到这个 RMWebApp 中。配置文件中有个配置项“yarn.web-proxy.address”,就是代码中的 PROXY_ADDRESS,这是“代理服务器”的地址。如果这个地址与另一个配置项“yarn.resourcemanager.webapp.address”,即 RM_WEBAPP_ADDRESS 所给定的地址(IP 地址加端口号)相符,或者不作规定,那就说明这个代理服务器与 RMWebApp 可以合二为一,把 WebAppProxyServlet 注入 RMWebApp。如果不相符呢?那就说明选择了另开一个代理服务器 WebAppProxyServer,通常这还是在 RM 所在的节点上,只是要另起一个 JVM,专门运行这 WebAppProxyServer,但是也可以不在 RM 上。

完成了准备之后,就创建 RMWebApp 对象,然后由 Builder.start() 启动这个 WebApp。那么 RMWebApp 是个线程吗?不是的,RMWebApp 本身并非线程;但是 RMWebApp 内部,实际上凡是 WebApp 类对象内部,都有个 HttpServer2 对象,而 HttpServer2 内部又有个 Server 类对象 webServer,那才是线程。这里的 HttpServer2 类是对 HttpServer 类的改进,是 Hadoop 的 2.2 版本才有的,以前是 HttpServer。不过 HttpServer2 与 HttpServer 并无继承

和扩充的关系。为保持兼容,Hadoop 的代码中仍保留了原先定义的 `HttpServer`,但以后就都使用 `HttpServer2` 了。二者内部都有个成分为 `Server` 类对象,这是个线程,但请注意这个 `Server` 可不是我们 RPC 层的那个 `Server`,这个 `Server` 类是 `org.mortbay.jetty.Server`,是从第三方的开源软件 `Jetty` 导入的。Hadoop 本身的代码中没有这个类的定义,那是在 `Jetty` 的代码中,我们只是使用它,这里就不深入到 `Jetty` 里面去了。我们只需知道,虽然实际的 Web 服务线程是用的第三方提供的 `Servlet` 引擎,但是这个引擎会回调我们替它绑定的 `Servlet`,即 `WebAppProxyServlet`,这就可以了。抽象类 `WebApp` 的摘要是这样的:

```
abstract class WebApp extends ServletModule {}
] enum HTTP { GET, POST, HEAD, PUT, DELETE }
] HttpServer2 httpServer
] Server webServer //这是 org.mortbay.jetty.Server,是个线程
] GuiceFilter guiceFilter
] Router router //根据 URL 找到相应的对象、函数、页面
] TreeMap<String, Dest> routes = Maps.newTreeMap(); // path->dest
] find(Class<T> cls, String cname)
] lookupRoute(WebApp.HTTP method, String path)
] resolve(String httpMethod, String path) //Resolve a path to a destination.
] resolveAction(WebApp.HTTP method, Dest dest, String path)
] route(HTTP method, String pathSpec, Class<?extends Controller> cls, String action)
```

正因为服务线程是在 `HttpServer2` 里面,在创建和启动 `WebApp` 的过程中最核心的操作就是 `HttpServer2` 对象的创建,然而那并不是在 `RMWebApp` 的构造方法中,而是在 `builder.start()` 的过程中:

```
[ResourceManager.main() > ResourceManager()
=> ResourceManager.serviceStart() > startWebApp() > WebApps.Builder.start()]

WebApps.Builder.start(WebApp webapp) //webapp 的实参是个 RMWebApp
> webapp.setName(name) //“cluster”,见前面的 WebApps.$for()
> webapp.setWebServices(wsName) //将这 WebApp 的名称设置成“ws”
> String basePath = "/" + name //“/cluster”
> webapp.setRedirectPath(basePath) //设置重定向路径
>> this.redirectPath = path
> ... //关于路径和 URL 的种种设置,此处从略
> if (this.httpPolicy == null) { //采用 HTTP 还是 HTTPS?
>+ String httpScheme = WebAppUtils.getHttpSchemePrefix(conf) //未作规定就默认 HTTP
> } else { //有规定就按规定办,或 HTTPS,或 HTTP
>+ String httpScheme = (httpPolicy == Policy.HTTPS_ONLY)?
WebAppUtils.HTTPS_PREFIX : WebAppUtils.HTTP_PREFIX
> }
> HttpServer2.Builder builder = new HttpServer2.Builder().setName(name)
```



```

        .addEndpoint(URI.create(httpScheme + bindAddress + ":" + port))
        .setConf(conf).setFindPort(findPort)
        .setACL(new AdminACLsManager(conf).getAdminAcl())
        .setPathSpec(pathList.toArray(new String[0]))
        //创建 HttpServer2.Builder 对象,那是 HttpServer2 对象的构建者
> boolean hasSpnegoConf = spnegoPrincipalKey!= null &&
    conf.get(spnegoPrincipalKey)!= null &&
    spnegoKeytabKey!= null && conf.get(spnegoKeytabKey)!= null
    //Spnego 是一种 Client/Server 双方关于身份认证方法的协商机制
> if (hasSpnegoConf) {
    //把 UGI 中关于是否采用安全机制的规定延伸到该用户的 kerberos 机制中
>+ builder.setUsernameConfKey(spnegoPrincipalKey).setKeytabConfKey(spnegoKeytabKey)
    .setSecurityEnabled(UserGroupInformation.isSecurityEnabled())
> }
> if (httpScheme.equals(WebAppUtils.HTTPS_PREFIX)) {
>+ WebAppUtils.loadSslConfiguration(builder) //如果采用 HTTPS 就要装载 SSL 的配置
> }
> server = builder.build() == HttpServer2.Builder.build() //创建 HttpServer2 对象
>> HttpServer2 server = new HttpServer2(this)
>>> String appDir = getWebAppsPath(b.name)
>>> this.webServer = new Server() //创建服务线程,即 org.mortbay.jetty.Server
>>> initializeWebServer(b.name, b.hostName, b.conf, b.pathSpecs)
>>>> maxThreads = conf.getInt(HTTP_MAX_THREADS, -1)
>>>> QueuedThreadPool threadPool = maxThreads == -1?
    new QueuedThreadPool() : new QueuedThreadPool(maxThreads) //创建工作线程池
>>>> webServer.setThreadPool(threadPool)
>>>> ...
>>>> webServer.addHandler(webAppContext)
>>>> addDefaultApps(contexts, appDir, conf)
>>>> addGlobalFilter("safety", QuotingInputFilter.class.getName(), null)
>>>> addDefaultServlets() //添加下列默认 servlet,这些是每个 Web 服务器都有的
>>>>> addServlet("stacks", "/stacks", StackServlet.class)
>>>>> addServlet("logLevel", "/logLevel", LogLevel.Servlet.class)
>>>>> addServlet("metrics", "/metrics", MetricsServlet.class)
>>>>> addServlet("jmx", "/jmx", JMXJsonServlet.class)
>>>>> addServlet("conf", "/conf", ConfServlet.class)
>> if (this.securityEnabled) {
>>+ server.initSpnego(conf, hostName, usernameConfKey, keytabConfKey)
>> }
>> if (connector!= null) {

```

```

>>+ server.addUnmanagedListener(connector)
>> }
>> for (URI ep : endpoints) {
>>+ String scheme = ep.getScheme()
>>+ if ("http".equals(scheme)) { //采用 HTTP
>>++ listener = HttpServer2.createDefaultChannelConnector()
>>+ } else if ("https".equals(scheme)){ //采用 HTTPS
>>++ SslSocketConnector c = new SslSocketConnectorSecure() //添加 SSL 加密层
>>++ c.setNeedClientAuth(needsClientAuth)
>>++ c.setKeyPassword(keyPassword)
>>++ ...
>>++ listener = c
>>+ } //end if ("https".equals(scheme))
>>+ listener.setHost(ep.getHost())
>>+ listener.setPort(ep.getPort() == -1 ? 0 : ep.getPort())
>>+ server.addManagedListener(listener)
>> } //end for (URI ep : endpoints)
>> server.loadListeners()
>> return server
> for(ServletStruct struct: servlets) {
    //servlets 是前面 ResourceManager.startWebApp() 中通过 builder.withServlet() 设置好的
>+ server.addServlet(struct.name, struct.spec, struct.class) //一个个添加到 HttpServer2 中
    //前面实际上只设置了一个 WebAppProxyServlet, 见 ResourceManager.startWebApp()
    //按前面的设置, 这三个参数分别为“proxy”、“/proxy/*”和 WebAppProxyServlet.class
    //如果采用独立的代理服务器, 则 WebAppProxyServlet 也不在这个 servlets 集合中
> } //end for
> for(Map.Entry<String, Object> entry : attributes.entrySet()) {
>+ server.setAttribute(entry.getKey(), entry.getValue())
> }
> HttpServer2.defineFilter(server.getWebAppContext(),
    "guice", GuiceFilter.class.getName(), null, new String[] { "/" })
    //定义一个 GuiceFilter, 并与 URL 路径 “/” 绑定
    //GuiceFilter 是从 com.google.inject.servlet 引入的
> webapp.setConf(conf)
> webapp.setHttpServer(server) //将此 HttpServer2 对象设置成 WebApp 的 Web 服务器
> server.start() == HttpServer2.start() //启动 Web 服务器 HttpServer2
>> openListeners() //打开用于 Listen 操作的 socket
>> webServer.start() == Server.start()
    //启动 org.mortbay.jetty.Server, 这是在 HttpServer2 的构造函数中创建的
> LOG.info("Web app /" + name + " started at " + server.getConnectorAddress(0).getPort())

```

```

> module = new AbstractModule() //先创建一个动态扩充了 AbstractModule 的 module
    ] configure()
    > if (api != null) bind(api).toInstance(application)
> injector = Guice.createInjector(webapp, module) //再用此 module 创建一个 Guice Injector。
    //这个 Injector 会根据具体绑定实现各个依赖的注入
> webapp.setGuiceFilter(injector.getInstance(GuiceFilter.class))
    //安排用 GuiceFilter 进行 HTTP 请求的过滤
> return webapp

```

先说明一下这里对 `WebApp.setRedirectPath()` 的调用。人们在访问网站的时候,常常会光输入网址,或光是在网址后加一“/”。碰到这样的情况,服务器可能会将其“重定向”,引导到某个特定的网页,这里就是在设置重定向的目标。

然后这里的核心就是创建一个 `HttpServer2` 对象,为此先要创建一个 `HttpServer2` 的 Builder,即 `HttpServer2.Builder`,再由这个 Builder 构建 `HttpServer2` 对象,并往里面添加一系列的 servlet 及其名称和路径,这里面也包括前面 `ResourceManager.startWebApp()` 中通过 `builder.withServlet()` 加入到 `WebApps.servlets` 集合中的那些 servlet。不过在我们这个情景中那里面只有一个 servlet,就是 `WebAppProxyServlet`,路径为“/proxy/*”。

最后将这个 `HttpServer2` 对象设置成 `RMWebApp` 的 Web 服务器并加以启动。

这样,RM 在其初始的 `serviceStart()` 过程中创建了一个 `WebApp`,实际上是 `RMWebApp` 对象。而在这个 `RMWebApp` 的初始阶段,则创建了一个 `HttpServer2` 对象;并在后者的 `start()` 过程中又创建了一个从 `org.mortbay.jetty` 导入的 `Server` 类对象。同时,也替 `HttpServer2` 配备好了相应的各种 Filter 和 Servlet。Jetty 是个开源的 Http Server 或者说 Http Servlet 引擎。

当程序从 `WebApps.Builder.start()` 返回时,已经为 RM 完成了 Web 服务器的创建和初始化,这个服务器已经就绪,在等待来自客户浏览器的连接了。这个服务器内部有个线程池,每接受一个连接请求而建立了一个 Http 连接,便会从线程池中领用一个线程来处理这个连接上的服务请求,用完之后又归还给这个线程池。

这里涉及 Guice 和 Jetty 两个第三方软件包,前者主要是面向 Web 的 DI(Dependency Injection)框架,后者则是 Web 服务器和 Java Servlet 的容器。要深入到这两个软件包中就不免离题了(其实我也不甚了了),所以下面只就 Hadoop 代码所及做点简单介绍,有兴趣和需要的读者可以自行参阅相关的专著以及 Guice、Jetty 和 Tomcat 的源码。这三者都是开源的,前面二者依赖于 javax,然而 Sun 和 Oracle 只提供 javax 的 jar 文件而并不开源。开源的 OpenJDK 却并不包含这部分内容(javax 是对 JDK 的扩充),Tomcat 中倒是有 javax 的源码,但不知是否完整。

那么这个 Web 服务器是怎么跟客户交互并提供客户所请求的服务的呢? 我们又要回过头来看看 `RMWebApp` 的 `setup()`,这也是其初始化的一部分,这个 `setup()` 是在 `WebApp` 的 `configureServlets()` 中受到调用的。这里要先加深一下印象:`RMWebApp` 扩充了 `WebApp`,`WebApp` 提供了 `configureServlets()`,而 `RMWebApp` 提供了 `setup()`:

```

[ResourceManager.main() > ResourceManager() => ResourceManager.serviceStart()
> startWebApp() => WebApp.configureServlets()]

```

```

WebApp.configureServlets()    // configureServlets()是由 WebApp 提供的
> setup() == RMWebApp.setup()    //setup()是由 RMWebApp 提供的
>> bind(JAXBContextResolver.class) //JAXB 提供 Java 类与其 XML 表示之间的双向转换
>> bind(RMWebServices.class)    //绑定 RMWebServices 类,这是实际的服务提供者
                                //bind()是从 Inject 包中的 ServletModule 类继承的
>> bind(GenericExceptionHandler.class)
>> bind(RMWebApp.class).toInstance(this) //把 RMWebApp 类绑定到当前这个对象
>> if (rm!= null) { //如果给定了具体的 ResourceManager 对象 rm
>>+ bind(ResourceManager.class).toInstance(rm)
                                //将 ResourceManager 类绑定到这个特定的对象 rm
                                //程序中凡要用到 ResourceManager 的时候,就都用这个 rm
>>+ bind(RMContext.class).toInstance(rm.getRMContext())
                                //把 RMContext 类绑定到 rm.getRMContext()所返回的那个对象
>>+ bind(ApplicationACLsManager.class).toInstance(rm.getApplicationACLsManager())
                                // ApplicationACLsManager 针对具体的 Application,具体的作业
>>+ bind(QueueACLsManager.class).toInstance(rm.getQueueACLsManager())
                                // QueueACLsManager 针对作业调度队列
>> }
>> route("/", RmController.class)
>> route(pajoin("/nodes", NODE_STATE), RmController.class, "nodes")
                                //表示如果 URI 路径为"/nodes.node.state",就由 RmController.nodes()加以执行
>> route(pajoin("/apps", APP_STATE), RmController.class)
                                //把"/apps/:app.state"路由到 RmController.app()
>> route("/cluster", RmController.class, "about")
>> route(pajoin("/app", APPLICATION_ID), RmController.class, "app")
                                //把"/app/:app.id"路由到 RmController.app()
>> route("/scheduler", RmController.class, "scheduler")
>> route(pajoin("/queue", QUEUE_NAME), RmController.class, "queue")
                                //所有这些 URI 路径都导向 RmController,最后都要依靠 RmController 完成操作
> serve("/", "_stop").with(Dispatcher.class)
                                //告知 Web 服务底层,对于路径"/和"_stop",上层的服务由 Dispatcher 提供
> for (String path : this.servePathSpecs) { //包括 "/*"、"/BasePath/*"、"/wsName/*"等
>+ serve(path).with(Dispatcher.class) //上层对所有这些路径的服务都由 Dispatcher 提供
> }
> configureWebAppServlets() //将 RMWebAppFilter.class 绑定到"/*"这个 URI,详见后述

```

先要说明一下 WebApp 的这个 configureServlets()是如何受到调用的。WebApp 类的定义中提供了一个 configureServlets()方法,但是在 Hadoop 的源码中却找不到对此方法的调用。这是因为,抽象类 WebApp 是对 ServletModule 类的扩充,而后的代码是在 com.google.inject 这个包中,那是在 Google 的 Guice 软件中。Hadoop 源码中有关 Web 和

Servlet 的代码是要结合 Guice 的代码或文档才能真正搞明白的,但是那当然已经超出了本书的范围,所以有关这方面的内容在本书中只能大致讲一下。事实上,在 Guice 的代码中,ServletModule.configureServlets()是个空函数,是在 ServletModule.configure()中受到调用的。但是 WebApp,作为对 ServletModule 类的扩充,以自己的 configureServlets()覆盖了那个空函数,所以从 ServletModule.configure()就调用到了 WebApp.configureServlets()。那么这个 ServletModule.configure()又是在什么情况下受到调用的呢?其实在 Guice 的代码中我们可以看到,ServletModule 是对抽象类 AbstractModule 的扩充,而 AbstractModule 中有个 configure(Binder builder),在那里调用了不带调用参数的 configure(),那实际上就是调用了 ServletModule.configure();因为 AbstractModule 中的 configure()是个抽象方法,而 ServletModule 则实现了这个方法。注意,这两个函数虽然同名,但前者有个 Binder 类的参数,而后者没有调用参数。然而这个 configure(Binder builder)又是在什么情况下受到调用的呢?仍是在 Guice 的代码中,是在 Elements.install(Module module)中受到调用的,应该是在安装一个 Module 的时候就调用其 configure(Binder builder)函数;而 WebApp 正是对 ServletModule 的扩充,这就是个 Module。把这个调用路线写下来,就是:

```
[Elements.install(Module module)>AbstractModule.configure(Binder builder)
> ServletModule.configure() >WebApp.configureServlets()]
```

所以,虽然 WebApp.configureServlets()是因 startWebApp()所导致,但是如果要细考其受调用的路线,则对其直接的调用来自 Guice,这是到 Guice 的代码中绕了一圈。

回到 WebApp.configureServlets()的代码摘要,里面有两项主要的操作,其一是对 setup()的调用,其二是对 serve().with()的调用。WebApp.setup()是个抽象方法,具体的方法要由实际的、扩充了 WebApp 的类提供,在这里就是 RMWebApp。所以这里对 setup()的调用就是对 RMWebApp.setup()的调用,这里已展开。而 serve().with(),则类似于一种英语句型,二者总是连用,这两个函数也都是从 Guice 中的 ServletModule 继承过来的。前面 serve()的参数是一个或几个路径,例如“/”,函数返回一个 ServletKeyBindingBuilder 类对象;这是定义于 ServletModule 内部的一个类,而 with()则是由这个类提供的方法。二者合在一起,就表示:如果 Http 服务请求中的路径是“/”,或“/_stop”,或 servePathSpecs 中的任一路径,就由 Dispatcher 类的对象提供服务。

这里的 Dispatcher 类是对 HttpServlet 类的扩充,所以 Dispatcher 的对象同时也就是 HttpServlet 类对象。Dispatcher 类是 Hadoop 自己定义的,但 HttpServlet 是从 javax.servlet.http 这个包(package)中导入的。我没有找到 javax.servlet.http 正式的源码,但是在开源项目 Tomcat 的代码中有个 javax 子目录,其下一层又有 servlet 子目录,使我们可以窥见一二:

```
abstract class HttpServlet extends GenericServlet {}
] service(HttpServletRequest req, HttpServletResponse resp)
  > String method = req.getMethod()
  > if (method.equals(METHOD_GET)) { //如果报文中请求的方法是 GET
  >+ ...
  >+ doGet(req, resp)
```

```

> } else if (method.equals(METHOD_HEAD)) { //如果报文中请求的方法是 HEAD
>+ ...
>+ doHead(req, resp)
> } else... //如此等等
] doGet(HttpServletRequest req, HttpServletResponse resp)
] doHead(HttpServletRequest req, HttpServletResponse resp)
] doPost(HttpServletRequest req, HttpServletResponse resp)
] doPut(HttpServletRequest req, HttpServletResponse resp)
] doOptions(HttpServletRequest req, HttpServletResponse resp)
> Method[] methods = getAllDeclaredMethods(this.getClass())
> for (int i = 0; i < methods.length; i++) {
>+ Method m = methods[i]
>+ if (m.getName().equals("doGet")) {
>++ ALLOW_GET = true
>++ ALLOW_HEAD = true
>+ }
>+ if (m.getName().equals("doPost")) ALLOW_POST = true
>+ ...
> }
> ...

```

HttpServlet 是对 GenericServlet 的扩充。后者之所以称为 Generic,是因为它并不专门针对某个特定的规程(协议)。而 HttpServlet 则将其具体化成专门针对 HTTP,就是所谓的互联网协议。HttpServlet 针对 HTTP 协议的要求提供了 doGet()、doHead()、doPost()等操作方法,还提供了一个总的调用入口 service()。

这些函数的调用参数都一样,就是一个(在 Servlet 这一层上)代表着具体 HTTP 请求的 HttpServletRequest 对象 req,那其实就是具体 HTTP 请求报文的有效载荷;还有一个基本空白有待填写的 HTTP 响应报文,就是 HttpServletResponse 对象 rep。不过 HttpServlet 中的 doGet、doPost 等函数都是抽象函数,有待继承和扩充 HttpServlet 类的实体类加以落实。而这里的 Dispatcher,就是一个具体化了的、扩充了的 HttpServlet,所以 Dispatcher 有自己的 service(),也有自己的 doGet、doPost 等,那才是真正用来提供服务的一种具体的、落实的 HttpServlet。前面的 serve().with()调用语句,就是把 Dispatcher 与一些特定的 URL 路径挂上钩。但是“Dispatcher”这个名称就告诉我们,这个对象的主要工作是分发,而不是最终的服务提供者,但是凭什么分发呢?当然是凭 URL,或者其中的一部分,而此前 setup()里面的那些 route()调用就是在为此预做准备。至于同在 setup()里面被调用的 bind(),即绑定,则用于“依赖注入”,即 DI。

RMWebApp 本身并未提供 bind()和 route()这两个方法,这也是从 ServletModule 继承过来的。所以这里调用的 bind()就是 ServletModule.bind(),route()也是如此,这都是从 Google Guice 的 inject.servlet 包中导入的。

RMWebApp.setup()的代码中首先通过 ServletModule.bind()将这个 ServletModule 与

一些类即模块,加以绑定。这里解决的是所谓“依赖注入”即 DI 的问题,对于 DI 和 bind() 这个函数的作用后面还要另做介绍。

然后又通过 route() 在一个“路由器”即 Router 对象中配置好一些“路由”。有了这些路由,根据 HTTP 请求中所载的方法(例如 GET、POST)和 URI 路径就可以知道对此请求应该执行何种行动。以这里的路由“route("/cluster", RmController.class, "about")”为例,这说明如果 URI 路径中的最后一节是“/cluster”,就应该创建一个 RmController 类对象并调用其名为“about”的方法。我们摘要看一下 RmController 类的定义,就可发现它确实提供了这么个方法:

```
class RmController extends Controller {}
] RmController(RequestContext ctx)
  > super(ctx)
] index()
  > setTitle("Applications")
] about()    //见 route("/cluster", RmController.class, "about")
  > setTitle("About the Cluster")
  > render(AboutPage.class) //向客户端的浏览器发送 web 页面 AboutPage
] app()      //见 route(pajoin("/app", APPLICATION_ID), RmController.class, "app")
  > render(AppPage.class)  //向客户端的浏览器发送 web 页面 AppPage
] ...
```

做好了这些准备,才通过 serve().with() 告知 Web 服务的底层:对于种种具体的 URI 路径,应该把请求上交给哪一个 HttpServlet 加以处理,在我们这个情景中是 Dispatcher。显然,这里所指定的几种路径都与 RM 和 YARN 相关,因为这是 RM 节点上的 WebApp,是 RM 节点上的 Web 服务器。

读者也许会有疑问,这里通过 serve().with() 连接的是 Dispatcher.class,这是一个类,而不是一个具体的对象,这个 Web 服务下层该把 HTTP 请求提交给哪一个具体的 Dispatcher 对象呢? 首先,对于这里的 Dispatcher 这个类并不存在这个问题,因为 Hadoop 的代码中在 class Dispatcher 的定义前面有个 @Singleton 标注:

```
@Singleton
public class Dispatcher extends HttpServlet {}
```

凡是带有 @Singleton 标志的类,一台 Java 虚拟机上就只能有一个这样的对象,这是由 JVM 在运行时加以保证的。Java 程序经编译之后的代码都由 JVM 加以执行,对象的创建当然也不例外。要创建 Dispatcher 对象时,JVM 会发现这个类带有 @Singleton 标注,自会保证不让重复创建。所以,这里的代码将一些路径引导到 Dispatcher,并不排除别的地方将别的路径也同样引导到这同一个 Dispatcher 对象,这个 Dispatcher 就是 Web 服务上层的总的门户与枢纽。关于“标注”,即 Annotation,后面还要介绍。

其实,即使不是 Singleton,只说明类而不具体到某一个对象也未必就是问题。我们只要想一下线程池,那就是许多同类的线程并存,需要时只要抓到其中任何一个空闲着的线程就行。虽然这里所要路由或绑定的并非线程,但道理是一样的。HTTP 是个无状态的协议,客

户这一次访问中调用了某类对象的某个方法,一般而言并不会改变这个对象的任何状态,不会在这对象中留下任何残余,对下次访问没有影响,所以本来就不必关心究竟是由哪一个对象处理。

18.3 Dependency Inject 和 Annotation

回头再说前面代码摘要中的“绑定”。为此得要简单介绍一下 Google 的 inject 软件包和“依赖注入(Dependency Inject,缩写为 DI)”这一种机制。我们知道,软件执行的过程是有层次的,如果一个类 A 中的某个方法在执行过程中调用了类 B 中的某个方法,我们就认为 A 依赖于 B。但是,根据实际的需要和情景,这个 B 往往可以是许多不同的 B1、B2、B3……之一,而且究竟要用其中的哪一个在编译/连接时还无法确定,需要在运行时动态加以绑定。读者不妨设想一下转接电话的过程,电话交换机得要根据所拨打的电话号码确定这是固定电话的转接还是移动电话的转接,固定电话与移动电话是两个不同的类,转接的方法也很不一样,但它们又都是电话。同样的道理,动态绑定在 Web 操作中也是很频繁的。在 C 语言中,这样的动态绑定是通过设置函数指针来实现的,但是让普通的应用程序员使用函数指针确实容易出错而不太安全。Java 语言中没有函数指针的概念而采用对象赋值,但本质还是一样的,那就是动态绑定。可是要让应用程序员自己编写这样的绑定就麻烦了,那样的话应用程序员就必须了解有关每个具体绑定的细节。特别是,许多绑定还不仅仅是设置一下函数指针那么简单,可能还得设置一些配套的调用条件和参数,那就更麻烦了。显然,要是能把这样的绑定操作封装起来,让应用程序员很方便地就能完成这些操作,当然是很有意义的。Google 的 Inject 软件包就实现了这样的机制,为应用程序员提供了方便。不过,Inject 并非独立的软件,而是专门针对 Web 应用的 Guice 库的一部分。如前所述,ServletModule 类就是从 com.google.inject 这个包中导入的。

在所绑定的这些类中,我们仍以 RMWebServices 为例特别加以关注。与我们平时所见的代码相比,RMWebServices 类的代码看上去有点特殊,下面是其中的若干片段:

```
@Singleton
@Path("/ws/v1/cluster") //注意 RMWebServices 的这个标注
public class RMWebServices {
    ...
    @GET
    @Path("/info")
    @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    public ClusterInfo getClusterInfo() {
        init();
        return new ClusterInfo(this.rm);
    }

    @GET
    @Path("/metrics")
```

```
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
public ClusterMetricsInfo getClusterMetricsInfo() {
    init();
    return new ClusterMetricsInfo(this.rm, this.rm.getRMContext());
}
```

```
...
}
```

这里的 `getClusterInfo()` 和 `getClusterMetricsInfo()` 是由 `RMWebServices` 提供的两个函数, 用来提供两种服务, 其 Java 代码并没有什么特别之处, 只是在函数的代码之前有一些以字符 `@` 开头的“标注(Annotation)”。

标注是 Java 语言自 Jdk1.5 开始才提供的一种机制, 这个机制让程序员可以为 class 定义、class 内部成分、方法、调用参数等程序元素加上各种特殊的注解。标注的用途是多方面的: 有些标注是对于编译的要求和指示, 让编译器执行一些特殊的检验, 有点像类型检查和程序中的断言(assert)检验; 有些标注则要求编译器自动生成一些代码, 有点像宏操作定义; 还有些标注是针对 JVM 的, 要求 Java 虚拟机在运行时执行到某些关键点时动态插入一些过程调用。不过, 加不加标注并不影响所依附的代码本身, 这些 Java 代码经编译后所产生的指令序列保持不变, 即使要自动生成一些代码也是生成在另外一个附加的文件中。编译带有标注的代码时, 先要用一个工具 apt 加以预处理, 然后才是常规的 Java 编译, 不过较新版本的 Java 编译器已经把 apt 合并进去了(apt 是“Annotation Processing Tool”的缩写)。

其实 Hadoop 一般的代码中也有标注, 只是不太重要而没有受到我们的重视。例如, `@Override` 就表示一个函数是用来覆盖其基类(受到扩充的那个类)中的同名函数, 因而编译器应该检验在其基类的定义中确实有这样的函数(同样的函数名、同样的参数表), 否则就要提示出错。但是这对于我们阅读和理解代码并没有多大影响, 所以我们就忽略了。然而在涉及 Web 服务的代码中, 例如在 `RMWebServices` 类的代码中, 情况就不同了。这里的 `@GET` 标注表示所标注的函数, 即 `getClusterInfo()` 和 `getClusterMetricsInfo()`, 是在接收到来自客户端的 HTTP GET 操作请求时受到调用的。两个 `@Path` 标注, 则表示当 GET 操作所针对的 URI 是 `/info` 时就调用 `getClusterInfo()`, 是 `/metrics` 时则调用 `getClusterMetricsInfo()`。注意, 这里的 `/info` 和 `/metrics` 都是指 URI 中网址或域名之后的那一段, 类似于相对路径名, 例如 `/info` 实际上可以是 `http://www.mywebsite.com/info`, 也可以是 `127.0.0.0:8080/info`。至于两个 `@Produces`, 则表示函数返回的 `ClusterInfo` 和 `ClusterMetricsInfo` 对象在载入 HTTP 响应报文时应转换成 JSON 格式或 XML 格式。

标注分“元标注(Mete Annotation)”和普通标注两种, 其中元标注用于普通标注的定义和实现, 是关于标注的标注。普通标注则又分“固有(Built-in)”和“自定义(Customer)”两种, 前者是 Java 语言所固有的, 例如 `@Override`、`@Deprecated`、`@SuppressWarnings`, 凡是需要加标注的代码前面都要导入 `java.lang.annotation.Annotation`; 后者即自定义标注, 则需要由程序员加以定义和实现, 例如 `@Path` 就是个自定义标注。

对于自定义标注, 程序代码中的某处一定要有个类似于这样的定义:

```
public @interface Path {
```

```
...
```

```
}
```

这说明自定义标注相当于界面定义,但又是特殊的界面定义,所以在关键字 `interface` 前面要加上 `@` 符号。

读者也许要问,这些标注并非 Java 语言提供的元标注,那么就一定是在什么地方定义的,它们的定义在哪里呢?在 Hadoop 的代码中?当然,要在 Hadoop 的代码中定义也是可以的,事实上也确实定义了一些;但实际上 Hadoop 也从一些开发包模块中导入了现成的定义。以上面看到的 `@Singleton` 标注的定义为例, `RMWebServices.java` 中所 `import` 的就是 `com.google.inject.Singleton`。

对 Annotation 缺少了解的人往往会觉得这个东西很神秘,它的这些功能是怎么来的呢?我们不妨通过一个最简单的实例来了解一下。Hadoop 的代码中有一个自定义的标注 `DoNotPool`,这个标注的定义是这样:

```
public @interface DoNotPool {  
}
```

这是个界面,但是这个界面并不要求你提供任何方法,所以任何一个类只要愿意都可以声称实现了这个类,其实声不声称也没什么关系。但要是界面上定义了什么方法,那又另做别论。总之,这就是个界面定义。然而它的特殊性表现在:Java 语言规定,在每个类或方法(包括构造方法)的 `.class` 对象中,都有关于 Annotation 的结构成分,所以只要你在某个类或方法的定义前面加上了已经定义的标注,这个事实就会在编译的时候被记录在它的 `.class` 对象中。这个 `.class` 对象的内容主要当然是供 JVM 自用的,但是别忘了 Java 作为一种半解释性的语言还提供了“反射(reflection)”机制,使得在应用程序中也能看到这些信息(如果是 `public`)。事实上 Hadoop 的代码中就有这么一个类:

@DoNotPool

```
public class BuiltInGzipDecompressor implements Decompressor {  
    ... //我们并不关心这个类中定义了什么  
}
```

这样,经过编译以后,就会有 `BuiltInGzipDecompressor.class`,在它的数据结构中就会记载加了 `DoNotPool` 标注。当然,前面既然定义了 `DoNotPool` 这个界面,经编译以后就也会有个 `DoNotPool.class`。

于是,Hadoop 代码中的某处就可以有这样的语句:

```
if (decompressor.getClass().isAnnotationPresent(DoNotPool.class)) {  
    return;  
}
```

这是在检测 `decompressor` 这个对象所属的类,在其定义中是否带有 `DoNotPool` 这个标注,如果是就马上返回了。

但是这里有个问题,是谁提供了 `isAnnotationPresent()` 这个方法?可想而知, `getClass()` 这个方法返回的是个 `Class`,而 `isAnnotationPresent()` 这个方法显然是由 `Class` 类提供的。

由此可见,标注,即 Annotation,这个机制的功能是由几个方面里应外合实现的,我们平时

之所以觉得这东西似乎有点神秘,是因为我们看不到有关各方的代码。当然,我们在这里也还没有看到编译器中如何处理标注的代码,但那已经是不难加以合理推测的了。

18.4 对网页的访问

现在我们要回到前面的话题。

先看 RMWebServices 类的定义,那前面加了 3 个标注。第一个标注是 @Singleton,表示在同一台 Java 虚拟机上这个类的对象只能有一个,而不允许创建多个 RMWebServices 对象。第二个标注是 @Path("/ws/v1/cluster"),这是个带参数的标注,表示如果 Web 服务请求的 URI 路径中的节点“/ws/v1/cluster”对应着 RMWebServices,Web 服务器就应该把这个请求转到这儿来。如前所述,在 startWebApp()的过程中通过 RMWebApp.setup()绑定了 RMWebServices。这个“绑定”是通过调用 RMWebApp.bind()实现的,而 RMWebApp.bind()就是 ServletModule.bind(),那是从 Google 的 com.google.inject.servlet 导入的。

作为第三方软件 Google Inject 的使用者,Hadoop 只要在其 ServletModule 基础上加以扩充,定义自己的 WebApp 类并创建该类对象,再调用从 ServletModule 继承而来的 bind()、route()等方法就行,至于 Web 服务器对于 HTTP 报文的接收、检验和分发等等的底层处理,就不用管了,Jetty 和 Google Inject 自会提供这些功能,这样当然省了很多事。

不过光是把 RMWebServices 绑定到“/ws/v1/cluster”还不够,RMWebServices 是个类,它不是提供具体服务的操作,具体的操作是由“方法”即函数提供的,应该使用何种方法则须视具体的 HTTP 请求而定,我们继续看上面所引的 RMWebServices 类代码。里面定义了 getClusterInfo()和 getClusterMetricsInfo()这两个方法。二者前面都有 @GET 标注,表示这两个函数都是供 Web 服务器在接收到 HTTP GET 请求时加以调用的。但是两个函数的 @Path 标注带着不同的参数,一个是“/info”,另一个是“/metrics”。这表示,同是 HTTP GET,如果报文中 URI 所指定的目标是例如“localhost/ws/v1/cluster/info”,就调用 getClusterInfo(),以获取关于集群的一般信息;而如果目标是例如“localhost/ws/v1/cluster/metrics”,则调用 getClusterMetricsInfo(),以获取有关的统计信息。

有了这样的安排,如果用户发来 URI 为“.../ws/v1/cluster/metrics”的 HTTP GET 请求报文(不管是通过在浏览器的地址栏中输入这个 URI,还是点击某个网页中指向这个 URI 的链接,或者不用浏览器而只是以此为命令行参数使用工具软件 curl),底层的 Jetty 和 Guice/Inject 等软件在接收、解析该报文之后就会调用 RMWebServices.getClusterMetricsInfo(),Hadoop 只要在这个函数中提供如何获取有关统计信息的具体过程就行了。

不过这中间还有个缺口,还少了一个环节,这个缺口是由 Dispatcher 补上的。

回顾前面 WebApp.configureServlets()的代码摘要,里面有这么几个语句:

```
[ResourceManager.serviceStart()>startWebApp()=> WebApp.configureServlets()]
```

```
WebApp.configureServlets()
```

```
> setup()
```

```
> serve("/", "_stop").with(Dispatcher.class)
```

//告知 Web 服务底层,对于路径“/”和“_stop”,上层的服务由 Dispatcher 提供

```

> for (String path : this.servePathSpecs) {
>+ serve(path).with(Dispatcher.class)
    //上层对所有这些路径的服务都由 Dispatcher 提供,Dispatcher 是一个 HttpServlet
> }
> configureWebAppServlets()

```

Dispatcher 是对 HttpServlet 类的继承和扩充,后者来自 javax.servlet.http,是由 JDK 提供的。所以 Dispatcher 类的对象同时也是个 HttpServlet 类对象。凡是 HttpServlet 对象都有个方法 service(),Web 服务的底层就通过这个函数调用具体 Servlet 的服务,对于 WebApp 这就是 Dispatcher.service()。

上面代码中所调用的函数 serve(),实际上是 ServletModule.serve(),因为 WebApp 是对 ServletModule 的扩充,而 WebApp 并未定义自己的 serve()。事实上,与前面的 bind()一样,这也正是 Inject 所提供的使用界面的内容之一,Inject 软件的使用者通过 ServletModule.serve()向 Web 服务的底层登记:对于这里给定的每一个 URI 路径 path,还有“/”和“_stop”,都应该调用 Dispatcher.service()(其中“_stop”用于调试)。

所以,我们在这里看到的其实就是 Hadoop 与 Web 服务底层之间的分界。分界的上面,即 Hadoop 这一边,就是 Dispatcher.service(),以及 Dispatcher 背后的 Router 和由 RMWebServices 提供的针对特定 HTTP 请求的种种方法。这里我们只看到用于 GET 的方法,但是根据实际需要也可以有针对 POST、HEAD、PUT 和 DELETE 的方法。而分界的下面,则由 Inject 直接或间接提供(ServletModule 来自 com.google.inject.servlet)。

至于 WebApp 从 ServletModule 继承的 serve()、bind(),则提供了上下层之间的接插手手段。此外如@GET、@PATH 之类的标注在这里也起着重要的作用。从代码中看,前面通过 serve().with()把相关的 URL 路径导向 Dispatcher,让下层软件调用 Dispatcher.service();然后通过 bind()把服务的提供者绑定到 RMWebServices。又进一步通过 Router 把这些路径延伸导向 RmController。这些标注都是从 javax.ws 导入的,它们的界面定义都在 javax.ws 中。当然,Inject 也会从 javax.ws 导入这些标注的界面定义。如前所述,Java 的预处理工具 apt 或编译器会将这些标注转换成代码。

进一步,这里还调用了函数 configureWebAppServlets():

```

[ResourceManager.serviceStart()>startWepApp()=> WebApp.configureServlets()
> configureWebAppServlets()]

WebApp.configureWebAppServlets()
> if (this.wsName != null) { //我们知道这个 wsName 是“ws”,
    //见 ResourceManager.startWepApp()中对 WebApps.$for()的调用
>+ String regex = "(?!/" + this.wsName + ")"
>+ serveRegex(regex).with(DefaultWrapperServlet.class)
>+ Map<String, String> params = new HashMap<String, String>()
>+ params.put(ResourceConfig.FEATURE_IMPLICIT_VIEWABLES, "true")
>+ params.put(ServletContainer.FEATURE_FILTER_FORWARD_ON_404, "true")

```



```

>+ ...
>+ filter_class = getWebAppFilterClass()
>+> return RMWebAppFilter.class
>+ filter("/ * ").through(filter_class, params)
    //指定凡 URI 为"/ *"的 HTTP 请求均由 RMWebAppFilter 加以过滤
> }

```

这里的 `filter().through()` 与前面的 `serve().with()` 相似,后者是说什么样的页面路由由什么类提供服务,前者是说什么样的页面路由由什么类提供过滤。这个 `filter().through()` 调用表示凡网址后面为“/ *”的路径(实际上是全部)均由 `RMWebAppFilter` 加以过滤,就是调用 `RMWebAppFilter` 的 `doFilter()` 函数加以处理。下面我们将看到此种过滤的作用。

经过了这些铺排,就好像在一块接线板上把线都给连接好了,这块接线板就成了专用于 RM 的 Web 服务“接线板”,这个 Web 服务器就可以提供作为 RM 界面的 Web 服务了。

现在假设来了一个 HTTP GET 请求,其 IP 地址和端口号指向 RM 节点上的 Web 服务,其 URI 则为“/app/app.id”,这里的 *app.id* 是个具体的 AppID 字符串。这个 GET 请求和 URL 的来源可以有很多,比方说:用户通过浏览器在某个网页上点击某个链接;或者在浏览器的 URL 栏中直接输入路径;也可以不用浏览器而用命令行工具 `curl`;还可以在例如 Ruby 脚本中发出 HTTP GET 请求;甚至还可以在“爬虫”软件中自动发出这样的请求。

用户发出的也可以不是 HTTP GET 请求,而是 HTTPS GET 请求,二者的区别在于后者经过 SSL 加密,所以在 Web 服务这一端要加以解密。

Hadoop 这边 RM 节点上的 Web 服务底层接收到这么一个 HTTP GET 请求之后,首先是根据 URI 过滤,前面通过 `filter().through()` 规定了凡路径为“/ *”的都由 `RMWebAppFilter` 进行过滤,所以 `RMWebAppFilter.doFilter()` 就得到了调用:

```

RMWebAppFilter.doFilter()
> String uri = HtmlQuoting.quoteHtmlChars(request.getRequestURI())
> if (uri == null) uri = "/"    //如果 URL 只是一个网址,就在后面添上"/"
> RMWebApp rmWebApp = injector.getInstance(RMWebApp.class)
    //获取相应的 RMWebApp 对象
> rmWebApp.checkIfStandbyRM()  //检查这个 WebApp 是不是在 Standby 的 RM 中
>> standby = (rm.getRMContext().getHAServiceState() == HAServiceState.STANDBY)
> if (rmWebApp.isStandby() && shouldRedirect(rmWebApp, uri)){
    //如果这是在 StandbyRM 上,就把这 HTTP 请求重定向到 ActiveRM 上
>+ String redirectPath = rmWebApp.getRedirectPath() + uri
>+ if (redirectPath != null && !redirectPath.isEmpty()) {
>+ String redirectMsg =
    "This is standby RM. Redirecting to the current active RM: " + redirectPath
>+ response.addHeader("Refresh", "3; url = " + redirectPath)
    //发回客户端的是一个 Refresh 报头响应
    //使其延迟 3 秒钟后转向 ActiveRM 节点
>+ PrintWriter out = response.getWriter()

```

```

> ++ out.println(redirectMsg)
> ++ return //这里没有看到把响应报文发回客户端,但 Web 服务底层自会发送
> + }
> }
> super.doFilter(request, response, chain) == GuiceContainer.doFilter(request, response, chain)
//如果无需重定向就调用由 GuiceContainer 提供的 doFilter,那会调用相关的 servlet

```

在启用了 HA 机制的 Hadoop 集群中有 ActiveRM 和 StandbyRM 两个 RM 节点,二者都有 Web 服务器,但是实际的服务应该由 ActiveRM 的 RMWebApp 提供,因为这个节点上有更多的动态信息。所以,在 StandbyRM 节点上,如果事先有安排(设置了重定向地址)并满足 shouldRedirect()的条件,就向客户端发回一个 Refresh 报头,让客户端显示一行信息,等待 3 秒钟后就转向 ActiveRM 节点。Refresh 与 Redirect 相似,都有重定向的功能,所以这里实际上是在要求客户端另发一个 HTTP 请求,发到 ActiveRM 那里去。而若是在 ActiveRM 节点上,或者虽然在 StandbyRM 上但不满足 shouldRedirect()的条件,就无须这样的重定向,于是直接调用由 GuiceContainer 提供的 doFilter()。注意,RMWebAppFilter 是对 GuiceContainer 的扩充,所以它的 super.doFilter()就是 GuiceContainer.doFilter()。

那么这个 shouldRedirect()的条件又是怎样的呢?

```

RMWebAppFilter.shouldRedirect(RMWebApp rmWebApp, String uri)
> return !uri.equals("/") + rmWebApp.wsName() + "/v1/cluster/info") &&
    !uri.equals("/") + rmWebApp.name() + "/cluster") &&
    !NON_REDIRECTED_URIS.contains(uri)

```

就是说,如果既不是“/ws/v1/cluster/info”,也不是“/ws/v1/cluster”,并且也不在 NON_REDIRECTED_URIS 这个集合之内,如“/conf”、“/stacks”等,就符合了重定向的条件。反之如果 uri 恰好是“/ws/v1/cluster/info”或“/ws/v1/cluster”,就不会被重定向。

至于在 GuiceContainer.doFilter()中,则先前通过 serve().with()进行的设置就起作用了,实际上 RMWebApp 所支持的网页全都由 Dispatcher 提供服务。

经过这样的处理,根据这个 URI 和先前 serve().with()的设置确定了应该由 Dispatcher 这个 HttpServlet 加以处理,就调用其 service()方法,我们就从这个入口开始往下看:

```

Dispatcher.service(HttpServletRequest req, HttpServletResponse res)
//此时还只知道是 HTTP 请求,不知道究竟是 GET 还是 POST、PUT,还是别的
> res.setCharacterEncoding("UTF-8") //设置字符编码格式
> String uri = HtmlQuoting.quoteHtmlChars(req.getRequestURI()) //从报文中获取 URI
> if (uri == null) uri = "/" //URI 为空就默认为“/”
> if (uri.equals("/")) { //如果 URI 为“/”,就加以重定向
> + String redirectPath = webApp.getRedirectPath()
> +> return this.redirectPath //WebApp.redirectPath 是在 WebApps.start()中设置的
//来源于创建 Builder 时的参数或通过 setRedirectPath()设置
> + if (redirectPath != null && !redirectPath.isEmpty()) { //如果对此设置了重定向

```

```

> ++ res.sendRedirect(redirectPath) //则向客户端发送重定向报文,让客户端转向
> ++ return
> + }
> }
> //如果 URI 不是"/",或无须重定向
> String method = req.getMethod() //从请求报文中读取操作方法名称,例如 GET
> if (method.equals("OPTIONS")) { //再根据操作名称调用相应的方法
> + doOptions(req, res)
> + return
> }
> if (method.equals("TRACE")) {
> + doTrace(req, res)
> + return
> }
> if (method.equals("HEAD")) {
> + doGet(req, res) => HttpServlet.doGet(req, res); // default to bad request
> + return
> }
> //既不是 OPTIONS,也不是 TRACE 或 HEAD,那就是 GET 或 PUT、POST、DELETE
> String pathInfo = req.getPathInfo() //从请求报文中读取 URI
> if (pathInfo == null) {
> + pathInfo = "/" //没有给出路径就默认为"/"
> }
> Controller.RequestContext rc = injector.getInstance(Controller.RequestContext.class)
//获取一个 Controller.RequestContext 对象
> if (setCookieParams(rc, req) > 0) { //根据请求报文设置 Context 中的 Cookie
> + ...
> }
> //Cookie 没问题,继续往下走
> rc.prefix = webApp.name()
> Router.Dest dest = router.resolve(method, pathInfo)
== Router.resolve(method, pathInfo)
//根据报文中所载的 URI 和方法(此处假定为 GET)在路由器中寻找目标
> if (dest == null) { //找不到通向任何 Controller 的路由,返回 SC_NOT_FOUND
> + rc.setStatus(res.SC_NOT_FOUND) //设置状态为 NOT_FOUND
> + render(ErrorPage.class) //然后发送出错页面 ErrorPage
> + return
> }
> //根据 URI 找到了目标 dest
> setMoreParams(rc, pathInfo, dest)

```

```

> Controller controller = injector.getInstance(dest.controllerClass)
           //从路由器输出的 Dest 对象中获取具体的 Controller,此处为 RmController
> // TODO: support args converted from /path/:arg1/...
> dest.action.invoke(controller, (Object[]) null) == Method.invoke()
           //启动 Controller 中具体的 Action,例如 app()
>> RmController.app()
> if (!rc.rendered) {
>+ if (dest.defaultViewClass != null) {
>++ render(dest.defaultViewClass)
>+ } else if (rc.status == 200) {
>++ throw new IllegalStateException("No view rendered for 200")
>+ }
> }

```

前面 URI 路径为“/”时有可能会需要重定向。这取决于 WebApp 中的 `redirectPath` 是否有设置,那是可以通过 `WebApp.setRedirectPath()` 加以设置的。设置 `redirectPath` 的目的是登记一个网页或站点,即另一个 URL,把客户端引导到那个网页或站点上去。

不过我们这个情景中的 URI 并非“/”,因此程序会往下执行,直到通过路由找到 `RmController` 中的函数 `app()` 并加以调用:

```
[Dispatcher.service() > RmController.app()]
```

```

RmController.app() //见 route(pajoin("/app", APPLICATION_ID), RmController.class, "app")
> render(AppPage.class)
>> getInstance(cls).render() == AppBlock.render(Block.html)
>>> String aid=$(APPLICATION_ID) //获取 GET 报文 URL 中作为字符串的 AppID
>>> appID = Apps.toAppID(aid)
>>> RMContext context = getInstance(RMContext.class)
>>> RMApp rmApp = context.getRMApps().get(appID)
           //根据 appID 从 RMContextImpl 的 ConcurrentMap<ApplicationId, RMApp> 中
           //获取代表着这个 App 的 RMAppImpl 对象
>>> AppInfo app = new AppInfo(rmApp, true, WebAppUtils.getHttpSchemePrefix(conf))
           //创建一个 AppInfo 对象,这个类的构造方法中要从 RM 收集不少信息
>>> String remoteUser = request().getRemoteUser()
>>> callerUGI = UserGroupInformation.createRemoteUser(remoteUser)
>>> if (callerUGI != null
    && !(this.aclsManager.checkAccess(callerUGI,
    ApplicationAccessType.VIEW_APP, app.getUser(), appID)
    || this.queueACLSManager.checkAccess(callerUGI,
    QueueACL.ADMINISTER_QUEUE, app.getQueue())) {
>>>+ puts("You (User " + remoteUser + ") are not authorized to view application " + appID)

```

```

>>>+ return
>>> }
>>> setTitle(join("Application ", aid)) //把两个字符串连在一起
>>> RMAppMetrics appMetrics = rmApp.getRMAppMetrics()
    //读取关于该 App 的许多统计信息,如进行了多少次 Attempt,耗用了多少资源等
>>> attempt = rmApp.getCurrentAppAttempt() //获取当前正在进行的 RMAppAttempt
>>> RMAppAttemptMetrics attemptMetrics = attempt.getRMAppAttemptMetrics()
    == RMAppAttempt.getRMAppAttemptMetrics() //读取当前 AppAttempt 的统计信息
>>>... //后面还有很多,但此处从略。

```

RmController.app()唯一的目的在于结果的 render(),即把网页投送和表现给客户看,这里要投送的是 AppPage,就是用来显示有关 App 信息的网页。网页的生成是相当复杂而琐碎的事,里面既有信息来源的问题,又有好多格式方面的细节问题。我们当然不关心格式和美工方面的细节,而只关心信息的来源。这部分信息的处理主要是 AppBlock 的事,最后落在 AppBlock 的 render()方法,即 AppBlock.render()。注意,Hadoop 的代码中有两个 AppPage、两个 AppBlock 的类型定义,其中之一,就是我们现在正在看的,是为作为 ResourceManager 内部成分的 WebApp 而定义的,因而这个 AppBlock 是在 RM 内部,在同一个节点、同一台 Java 虚拟机上,要在 RM 内部收集有关一个 App 的各种统计信息当然不是难事。

再看假定来自客户端的 HTTP GET 请求中的 URI 为“/proxy/ws/v1/cluster”时的情景。至于客户端为什么会发来这样的请求,那可以有多种原因,可以是点击了某个链接的结果,也可以是重定向的结果,还可以是在键盘上输入了这么一个 URI。我们在前面看到,服务端对于这个路径的服务是由 WebAppProxyServlet 提供的,因为与这个 servlet 绑定的路径是“/proxy/*”。但是这个 servlet 可以就在 ResourceManager 内部的 RMWebApp 中,也可以是在一个独立的 WebAppProxyServer 中。我们在这里假定 WebAppProxyServlet 在 RMWebApp 中。把这种情景搞懂了,采用独立 WebAppProxyServer 也是大同小异,因为最后总是靠 WebAppProxyServlet 提供服务。

顺便重温一下,RMWebApp 是对 WebApp 的扩充,WebApp 内部的成分 httpServer 是个 HttpServer2。而 WebAppProxyServer 则是对 HttpServlet 的扩充。HttpServer2 与 HttpServlet 之间并无继承与扩充的关系,但是二者性质和功能相近,并且都实现了 FilterContainer 界面,因而都提供 addFilter()和 addGlobalFilter()这两个函数。

这样,当 RMWebApp 中的 HttpServer2 服务器在接收到 HTTP GET 报文时会根据 URL 在它的 Filter 链中比对寻找,发现 URL 开头的路径是“/proxy/ws/v1/cluster”就知道应该交由 WebAppProxyServlet 处理,于是就会调用它的 doGet()函数。

我们看一下 WebAppProxyServlet 这个类的摘要:

```

class WebAppProxyServlet extends HttpServlet {}
] class Page extends Hamlet {}
]] Page(PrintWriter out)
    > super(out, 0, false)
]] html()

```

```

    > return new HTML<WebAppProxyServlet._>("html", null, EnumSet.of(EOpt.ENDTAG))
] notFound(HttpServletResponse resp, String message)
] makeCheckCookie(ApplicationId id, boolean isSet)
] doGet(HttpServletRequest req, HttpServletResponse resp) // HttpServer2 会调用 doGet()
    > ...
    > applicationReport = getApplicationReport(id)
    > ...
] getApplicationReport(ApplicationId id)
    > fetcher = (AppReportFetcher) getServletContext()
        .getAttribute(WebAppProxy.FETCHER_ATTRIBUTE))
        //返回预设的 AppReportFetcher 对象
        //见前面 ResourceManager.startWebApp() 中的 builder.withAttribute()
    > return (fetcher.getApplicationReport(id)
        == AppReportFetcher.getApplicationReport(id))

```

作为一个 servlet, WebAppProxyServlet 对于 GET 报文的处理和响应主要在于生成 Web 页面,但是页面中的内容要靠配套的 Fetcher 即 AppReportFetcher 去抓取。我们并不关心页面的格局和细节,我们关心的是页面上这些信息的来源。所以,我们在这里只关心 AppReportFetcher:

```

class AppReportFetcher {}
] AppReportFetcher(Configuration conf) //构造方法
    > this.conf = conf
    > applicationsManager =
        ClientRMProxy.createRMProxy(conf, ApplicationClientProtocol.class)
        //建立与 RM 节点的连接,并创建代表着 RM 的 proxy
] getApplicationReport(ApplicationId appId)
    > request = recordFactory.newRecordInstance(GetApplicationReportRequest.class)
    > request.setApplicationId(appId)
    > response = applicationsManager.getApplicationReport(request)
        //对 RM 节点上 applicationsManager 的 RPC 调用
    > return response.getApplicationReport() //返回 RPC 调用 getApplicationReport() 的结果

```

这些操作都来自 WebAppProxyServlet,我们知道 WebAppProxyServlet 的所在有两种可能:注入在 RMWebApp 内的 HttpServer2 中,在一独立的代理服务器 WebAppProxyServer 中。而 WebAppProxyServer,则可以与 RMWebApp 同在 ActiveRM 节点上(但在不同的 JVM 中),也可以在 StandbyRM 节点上。不管怎样,通过 RPC 与 RM 通信总是可以的,所以这里就通过 AppReportFetcher 对 RM 节点进行 RPC 调用。这里 applicationsManager 的类型为 ApplicationClientProtocol,说明在代理服务器与 RM 之间的通信协议是 ApplicationClientProtocol,但是实际上 applicationsManager 是代理服务器与下层 PB 和 RPC 机制之间的中介,对于 WebAppProxyServlet 而言这就是 RM 派驻在本地的 proxy。

RM 节点上与其对应的成分是 ClientRMService:


```

ClientRMService.getApplicationReport(GetApplicationReportRequest request)
> applicationId = request.getApplicationId()
> UserGroupInformation callerUGI = UserGroupInformation.getCurrentUser()
> RMApp application = this.rmContext.getRMAppls().get(applicationId)
> boolean allowAccess = checkAccess(callerUGI, application.getUser(),
                                     ApplicationAccessType.VIEW_APP, application)

> ApplicationReport report =
    application.createAndGetApplicationReport(callerUGI.getUserName(), allowAccess)
    == RMAppImpl.createAndGetApplicationReport(
                                     String clientUserName, boolean allowAccess)

> response = recordFactory.newRecordInstance(GetApplicationReportResponse.class)
> response.setApplicationReport(report)
> return response

```

这里由 RMAppImpl.createAndGetApplicationReport() 收集信息并创建的报告与前面在 AppBlock.render() 中收集并作为页面投送的信息大致相同。

理解了 ResourceManager 的 Web 服务, 要理解 NodeManager 的 Web 服务也就不难了。NodeManager 的 Web 服务与 ResourceManager 的略有不同, 这里另行定义了一个 WebServer 类。不过从大的方面看也仅此而已, 再怎么变, 大的格局还是一样的。

NodeManager 在其初始化过程中通过 createWebServer() 创建一个 WebServer 类对象 webServer:

```

[NodeManager.main() => NodeManager.serviceInit() > createWebServer()]

createWebServer(Context nmContext, ResourceView resourceView,
                 ApplicationACLsManager aclsManager, LocalDirsHandlerService dirsHandler)
> return new WebServer(nmContext, resourceView, aclsManager, dirsHandler)

```

作为一种 Service, WebServer 的内部有个 WebApp 类的对象 webApp。我们在前面见过 WebApp, 那是对来自 com.google.inject.servlet 的 ServletModule 的扩充, 但是加上了 HttpServer2 和 Router, 其中 HttpServer2 是线程。不过 WebApp 仍是抽象类, 所以在 RM 上扩充成了 RMWebApp, 在 NM 上则扩充成 NMWebApp。

所以, NM 节点上 WebServer 内部的 NMWebApp 与 RM 节点上的 RMWebApp 都是 ServletModule, 并且都是 WebApp, 二者的内部都有 HttpServer2 和 Router。知道了这一点, 下面就容易理解了。

我们看看 WebServer 和 NMWebApp 的摘要:

```

class WebServer extends AbstractService {}

] NMWebApp nmWebApp      // 只起临时变量的作用
] WebApp webApp          // 实际上是完成了初始化设置后的 NMWebApp
] WebServer(Context nmContext, ResourceView resourceView,

```

```

ApplicationACLsManager aclsManager, ...)

> super(WebServer.class.getName())
> this.nmContext = nmContext
> this.nmWebApp = new NMWebApp(resourceView, aclsManager, dirsHandler)
// 创建 NMWebApp 对象

] serviceStart()
> String bindAddress = WebAppUtils.getWebAppBindURL(getConfig(),
    YarnConfiguration.NM_BIND_HOST,
    WebAppUtils.getNMWebAppURLWithoutScheme(getConfig()))
> LOG.info("Instantiating NMWebApp at " + bindAddress)
> this.webApp = WebApps.$for("node", Context.class, this.nmContext, "ws")
    .at(bindAddress).with(getConfig())
    .withHttpSpnegoPrincipalKey(
        YarnConfiguration.NM_WEBAPP_SPNEGO_USER_NAME_KEY)
    .withHttpSpnegoKeytabKey(
        YarnConfiguration.NM_WEBAPP_SPNEGO_KEYTAB_FILE_KEY)
    .start(this.nmWebApp)
> this.port = this.webApp.httpServer().getConnectorAddress(0).getPort()

] class NMWebApp extends WebApp implements YarnWebParams{} // 定义于 Web Server 内部
]] ResourceView resourceView
]] ApplicationACLsManager aclsManager
]] LocalDirsHandlerService dirsHandler
]] NMWebApp(ResourceView resourceView, ApplicationACLsManager aclsManager,
    LocalDirsHandlerService dirsHandler)

> this.aclsManager = aclsManager
> this.dirsHandler = dirsHandler

```

从摘要中可以看到,在创建 WebServer 对象的时候,在其构造函数中创建了一个 NMWebApp,这虽说是 WebApp 的扩充,但是比 WebApp 只多了 resourceView 等几个数据成分。同是对 WebApp 的扩充,但 RMWebApp 和 NMWebApp 所扩充的内容有所不同。同样,刚创建时的 NMWebApp,其 WebApp 那一部分的有些成分是未加设置的,所以在 WebServer.serviceStart()中也要借助 WebApps.Builder 加以设置并启动。读者不妨对照前面对于 RMWebApp 的设置和启动自己阅读代码,这里就不加详述了。

再看 HDFS 的 NameNode。NameNode 内部的 httpServer 是个 NameNodeHttpServer 类对象,其内部也有个 HttpServer2 类对象 httpServer;而 HttpServer2 内部则有个 Server 类对象 webServer。这个 Server 类是从 Jetty 导入的 org.mortbay.jetty.Server,所以同样也是基于 Jetty 的。Jetty 是用 Java 语言编写的开源 Http Server,也就是 Web Server;从结构上说,这是一种 Servlet 容器,或者说 Servlet 池,也可以说是 Servlet 引擎。

最后,DataNode 内部的成分 infoServer 也是个 HttpServer2 对象,所以也是 Jetty Server,

也是 Servlet 容器。

ResourceManager 和 NodeManager 都属于 YARN 子系统,其用户界面相对比较简单,因为用户的操作也比较简单,除提交作业之外主要就是了解作业的进度。但是属于 HDFS 子系统的 NameNode 和 DataNode 就不同了,用户需要通过这个界面进行许多的文件操作,例如列目录、创建目录、拷贝文件、符号连接乃至 fsck、balance 等许许多多涉及文件系统的操作,所以理应有更多的 Servlet,让它们各司其职。

有关 Web 服务这一块是 Hadoop 代码中对第三方软件依赖最大的,代码中导入的 package 至少就有 org.jboss.netty、org.mortbay.jetty、com.sun.jersey、com.google.inject,这还没有算上 javax.servlet。所以,真要彻底把这一摊搞个明白,就得结合这些第三方软件,参考有关专著和资料。但是那显然已经超出了本书的范围而只好从略,有兴趣或需要的读者不妨自己加以研究。

第19章

Hadoop 的部署和启动

19.1 Hadoop 的运维脚本

系统的安装部署本来就不是小事,对于大规模的集群就更不用说了。Hadoop 一般都是在集群上运行,但是要运维人员跑到每一台机器上去部署或启动却是不现实的,得要能在一个集中的控制台节点上完成 Hadoop 的部署和启动(还有关机)才好,这当然又会使整个过程增加许多技术上的复杂度。既然是在一个集中的控制台节点上部署和启动一个集群,那当然就离不开远程操作,所以 Linux 的远程操作工具 ssh 和 rsync 就成了整个过程的基石。之所以是 ssh 和 rsync,而不是别的远程操作工具(比方说 Telnet),是因为这二者的安全性比较好,通信中采用了较强的加密手段。

Hadoop 的源代码中为此提供了一系列的脚本文件,供系统部署和启动之用。我们从脚本 start-all.sh 开始,就因为这是“start all”;尽管脚本中建议改用 start-dfs.sh 和 start-yarn.sh,对两个子系统分别加以部署和启动,但是为加深对过程的理解,我们宁可 from start-all 开始:

```
#!/usr/bin/env bash
# Start all hadoop daemons. Run this on master node.
echo "This script is Deprecated. Instead use start - dfs.sh and start - yarn.sh"
bin='dirname "${BASH_SOURCE-$0}"' # 用 dirname 命令构筑一个目录名,赋给变量 bin
bin='cd "$bin"; pwd' # 转入这个目录,并将操作 pwd 的返回值赋给 bin
DEFAULT_LIBEXEC_DIR="$bin"/.../libexec # 创建变量 DEFAULT_LIBEXEC_DIR
HADOOP_LIBEXEC_DIR=${HADOOP_LIBEXEC_DIR:-$DEFAULT_LIBEXEC_DIR}

.$HADOOP_LIBEXEC_DIR/hadoop - config.sh # 在当前进程内执行 hadoop - config.sh
# start hdfs daemons if hdfs is present
if [ -f "${HADOOP_HDFS_HOME}"/sbin/start - dfs.sh ]; then
    "${HADOOP_HDFS_HOME}"/sbin/start - dfs.sh -- config $HADOOP_CONF_DIR
fi # 若脚本文件 start - dfs.sh 存在就执行,部署和启动 Hdfs 在先
# 变量 HADOOP_CONF_DIR 来自 hadoop - config.sh

# start yarn daemons if yarn is present
if [ -f "${HADOOP_YARN_HOME}"/sbin/start - yarn.sh ]; then
    "${HADOOP_YARN_HOME}"/sbin/start - yarn.sh -- config $HADOOP_CONF_DIR
```

```
fi      #如脚本文件 start-yarn.sh 存在就执行,部署和启动 Yarn 在后  
      #变量 HADOOP_CONF_DIR 来自 hadoop-config.sh。
```

文件的第一行表明这是个 bash 语言的脚本,这种脚本一般都带有后缀名.sh,但是也可以不带。对 bash 缺少基本了解的读者可以参考一下有关的文档,此处不详细介绍。在 bash 语言中,从字符#开始直到本行结束为注解,这里的中文注解都是作者所加。

第一个命令行 echo 显示一行文字信息,说这个脚本已经过时,建议直接使用 start-dfs.sh 和 start-yarn.sh,但是实际上有个像 start-all 这样的总脚本也没有什么不好,还更加方便。

注意这里启动执行脚本 hadoop-config.sh 的那一行,其前面有个点号“.”,表示这个命令行是在当前进程,即 start-all.sh 的上下文中执行的,而不是另起一个进程加以执行,因而在执行过程中所创建的变量都在本进程中,可以继续使用。例如后面在执行 start-dfs.sh 的那个命令行中用到的变量 HADOOP_CONF_DIR,就是在 hadoop-config.sh 中创建的。如果没有前面的这个点号,像后面执行 start-dfs.sh 和 start-yarn.sh 那样,Bash 就会另行创建一个子进程来执行这个命令行,并等待子进程运行结束后再继续往下执行当前进程。如果是那样,子进程中创建的变量就不能在父进程中使用了。脚本 hadoop-config.sh 的作用在很大程度上就是创建一些变量,使父进程 start-all.sh 可以引用这些变量。

实际上在执行 start-dfs.sh 和 start-yarn.sh 的过程中还会执行好多脚本,而且那些脚本一般都比 start-all.sh 大得多。要在本章中列出每个脚本的内容,这篇篇幅就太大了。而且也不仅是篇幅大的问题,更重要的是那样很容易使一些重要的、实质性的核心操作淹没在许多细节中,而且也不容易让人看到操作的层次。所以,我们仍旧仿照对于 Hadoop 源代码的摘要处理,将脚本中那些重要的内容抽取出来形成摘要。

于是脚本 start-all.sh 的第一层摘要就是这样:

```
common-project/hadoop-common/src/main/bin/start-all.sh  
> hdfs-project/hadoop-hdfs/src/main/bin/start-dfs.sh  
> yarn-project/yarn/bin/start-yarn.sh
```

这里,我们把 hadoop-config.sh 的执行和另一些细节都省略了,以突出 start-dfs.sh 和 start-yarn.sh 这两个脚本的执行,使读者明白 start-all 的核心操作就是 start-dfs 和 start-yarn,并且是 start-dfs 在前、start-yarn 在后。然后,比如在 start-dfs.sh 下面,我们可以逐层展开和细化,以明了这个 start-dfs 的过程究竟是以什么样的次序执行了一些什么操作。

注意,本章所列脚本文件的路径都是各个脚本在 Hadoop 源码包中的路径,而不是安装以后实际用于命令行中的路径名。比方说,在脚本的命令行中 start-yarn.sh 运行时实际所在的位置是 YARN 分支上的 sbin 目录下,这是因为这个脚本被安装在那个目录中,但是它在 Hadoop 源代码包中的文件路径却是在 common-project/hadoop-common 分支上的 src/main/bin 目录下,列出这个路径可以方便读者阅读和分析。

下面,我们先假定 Hadoop 的软件尚未在集群上部署,看 start-all 这个脚本是怎样一步步把 Hadoop 部署到集群上并加以启动的。我们假定系统管理员已经在一个准备用作 ResourceManager 的主节点上登录,并在集群内的各个节点机上都有作为管理员的访问权限。至于 Hadoop 软件映像则可以在一台文件服务器上,也可以就在这个主节点上。说是准备用作 ResourceManager 的节点,并不排除也用它作为 NameNode。其实也可以把任何别的节点

用作“控制台”，而在控制台节点上启动安装，只是需要对个别脚本稍作修改。

19.2 Hadoop 的部署与启动

上面讲到的脚本 `start-all.sh` 把 Hadoop 的部署和启动综合在一起，运行这个脚本就既可完成 Hadoop 的部署又可启动其运行。如上所述，这个过程主要是两块，即脚本 `start-dfs.sh` 和 `start-yarn.sh` 的运行，而且 `start-dfs` 是在 `start-yarn` 之前。这很好理解，因为 YARN 是在 HDFS 的基础上运行的，当然先得把 HDFS 启动起来才能运行 YARN。另外这也表明，`start-dfs.sh` 和 `start-yarn.sh` 这两个脚本是把两个子系统的部署和启动综合在一起了。不过两个子系统的部署和启动其实很相似，理解了其中之一，另一个就不难理解了。所以，下面我们先着重讲解 HDFS 的部署和启动。注意，这里所列都只是摘要，实际上每个脚本都不是寥寥数行的事，而且每个脚本中都可能还执行了别的对于理解这个过程不太关键的脚本而这里并未列出。更重要的是，为便于理解，这里对一些比较隐晦难懂的语句做了解释、简化和改写，已经不是这些语句的原貌，真想把这个过程彻底搞懂的读者还是应该进一步直接去看原始的代码：

```
hdfs-project/hadoop-hdfs/src/main/bin/start-dfs.sh # 脚本 start-dfs
> common-project/hadoop-common/src/main/bin/hadoop-daemons.sh \
    -hostnames "$NAMENODES" --script "hdfs" start namenode
    # 先运行 hadoop-daemons 这个脚本
>> common-project/hadoop-common/src/main/bin/slaves.sh
    # 在 hadoop-daemons 中又先运行 slaves 这个脚本
>>> for each hostname; do
>>>+ ssh...
    # 通过 ssh 登录到每个 namenode 节点，以下操作就是在远端节点上执行的了：
>>>+ common-project/hadoop-common/src/main/bin/hadoop-daemon.sh # on remote node
>>>+> rsync -a -e ssh --delete --exclude = .svn --exclude = 'logs/*' \
    --exclude = 'contrib/hod/logs/*' $HADOOP_MASTER/ "$HADOOP_PREFIX"
    # 通过 rsync 从变量 HADOOP_MASTER 所指的来源拷贝 Hadoop 软件
>>>+> nohup nice -n $HADOOP_NICENESS $hadoopScript \
    --config $HADOOP_CONF_DIR $command "$@" &
    # 拷贝完成后即启动 namenode 子进程
    == hdfs-project/hadoop-hdfs/src/main/bin/hdfs start namenode
>>>+>>> exec "$JAVA" namenode
>>>+>>> echo $!>$pid
    # 将子进程的进程号记入 pid 文件
>>> done
- - - - 以上是第一遍执行 hadoop-daemons，是针对少数用作 NameNode 的节点 - - - -
- - - - 以下是第二遍执行 hadoop-daemons，是针对众多用作 DataNode 的节点 - - - -
> common-project/hadoop-common/src/main/bin/hadoop-daemons.sh \
    --script "hdfs" start datanode
>> common-project/hadoop-common/src/main/bin/slaves.sh
```



```

>>>> for each slave; do
>>>>+ ssh ... # ssh 到每个 datanode 节点,以下操作就是在每个远端节点上执行的了:
>>>>+ common-project/hadoop-common/src/main/bin/hadoop-daemon.sh # on remote node
>>>>+> rsync -a -e ssh --delete --exclude = .svn --exclude = 'logs/*'\
--exclude = 'contrib/hod/logs/*' $HADOOP_MASTER/ "$HADOOP_PREFIX"
# 也是通过 rsync 拷贝 Hadoop 软件
>>>>+> nohup nice -n $HADOOP_NICENESS $hadoopScript \
--config $HADOOP_CONF_DIR $command "$@" &
== hdfs-project/hadoop-hdfs/src/main/bin/hdfs start datanode
>>>>+>> exec "$JAVA" datanode
>>>>+>> echo $!> $pid # 将子进程的进程号记入 pid 文件
>>>> done

```

请特别注意这个摘要中的层次关系和时间顺序,并注意 `hadoop-daemons.sh` 和 `hadoop-daemon.sh` 是两个不同的脚本。

这个过程,从脚本中可见,是先后两次执行同一个脚本 `hadoop-daemons.sh`,而且两次都有同样的选项“`--script "hdfs"`”,表明两次执行 `hadoop-daemons.sh` 都有让它执行另一个脚本 `hdfs` 的目的。但是这两次执行所用命令行中的其他选项有所不同。首先前面一次带有选项“`-hostnames "$NAMENODES"`”这里的 `NAMENODES` 是个变量,实际上是个配置文件名,这个文件中列举了集群中用作 `NameNode` 的节点名,我们知道一个 Hadoop 集群中的 `NameNode` 节点可以不止一个。实际上,此次执行 `hadoop-daemons.sh` 的目的就是:把 Hadoop 软件部署到集群中安排用于 `NameNode` 的那些节点上并启动其运行 `NameNode`,所以另一个选项是“`start namenode`”。

相比之下,后面一次执行 `hadoop-daemons.sh`,其目的在于把 Hadoop 部署到集群中其余的那些节点上,并启动其运行 `DataNode`,所以另一个选项是“`start datanode`”,并且不带“`-hostnames`”选项。

为此,我们最好看一下 `hadoop-daemons.sh` 这个脚本,这里删去了一些并不紧扣主题但并非不重要的内容:

```

[start - all.sh > start - dfs.sh > hadoop-daemons.sh]

#!/usr/bin/env bash
usage = "Usage: hadoop-daemons.sh [--config confdir] \
[--hosts hostlistfile] [start|stop] command args..."
.$HADOOP_LIBEXEC_DIR/hadoop-config.sh
exec "$bin/slaves.sh" --config $HADOOP_CONF_DIR cd "$HADOOP_PREFIX" \
; "$bin/hadoop-daemon.sh" --config $HADOOP_CONF_DIR "$@"

```

这里也在当前进程的上下文中执行了 `hadoop-config.sh`,理由和作用同前述。

核心的操作是 `exec` 命令行,这个命令行启动执行脚本 `slaves.sh`,通过 `ssh` 逐一登录到每个 `slave` 节点上,并在对方节点上启动执行一个复合命令行,这个复合命令行包含着两个操作命令,中间以分号间隔。第一个操作是 `cd`,转入变量 `HADOOP_PREFIX` 所代表的目录;接着

的第二个操作是在那个目录中启动 `hadoop-daemon.sh` 的执行,这个命令的可选项“`$@`”代表着来自启动 `hadoop-daemons.sh` 这个脚本时的那些还没有被这个脚本用掉的选项和参数,都按原样下传,其中就包括“`--script "hdfs" namenode`”或“`--script "hdfs" datanode`”。

注意,这里所谓的 `slaves` 是相对于系统管理员所在的“主节点”而言的,与它们将来在 Hadoop 中的角色是主还是从并没有什么联系。事实上,当带有“`start namenode`”选项时,所针对的倒是将来 HDFS 的主节点。

于是,脚本 `hadoop-daemons.sh` 的执行引起了脚本 `slaves.sh` 的执行。至此为止的三个脚本,即 `start-all.sh`、`start-dfs.sh`、`hadoop-daemons.sh` 和将要运行的 `slaves.sh`,都是在系统管理员所在的“主节点”上运行的。

回到前面始于 `start-dfs.sh` 的分层摘要中,可以看到 `slaves.sh` 这一层的操作是:对于给定的每一个 `slave` 节点,即集群中有待部署和启动的节点,首先执行 `ssh`,远程登录到那个节点上,于是本机就变成了对方的一个远程终端。

由于 `slaves.sh` 是放在 `exec` 命令行执行的,同一命令行中对于 `hadoop-daemon.sh` 的启动就在 `slaves.sh` 的 `for` 循环中得到执行。后者的核心操作则是通过 `rsync` 从主节点拷贝 Hadoop 软件到本地,并执行脚本 `hdfs` 以启动 `namenode` 或 `datanode`。

这样综合起来看,`hadoop-daemons.sh` 的作用就是:对于每一个目标节点,先通过 `ssh` 操作登录到这个节点上,然后执行 `rsync` 和脚本 `hdfs`。

这就得说一下 `ssh` 和 `rsync` 的作用了。这个 `ssh` 是 Unix/Linux 的一个 Utility 工具软件,其作用是远程登录到另一台机器上,使本地的键盘和显示屏就好像连在对方机器上一样,建立起一个远程的 Shell 会话。或者,也可以说是使本地的终端变成对方的一个远程终端。早期的 Unix 上就有个这样的软件叫 `Telnet`,但是 `Telnet` 有不少问题,后来就有了 `rsh`,意为“Remote Shell”。可是 `rsh` 的安全性仍有问题,主要是传输内容没有加密,于是就又有了 `ssh`,意为“安全的(secure)`rsh`”。既然要在别的节点机上启动执行一些操作(`cd`, `rsync`,以及脚本 `hdfs`),那当然就得登录到对方的机器上。

而 `rsync`,则是个远程复制(同步)文件的工具软件。早期 Unix 上有个工具叫 `rcp`,意为“Remote copy”,也是因安全性有问题而为 `rsync` 所取代。

所以,这里使用的 `ssh` 和 `rsync` 都是有加密的,用 `ssh` 登录到对方还需要有 `password`,对方机器(宿主操作系统上)上还得有这么个账户,这些都得要事先加以准备。

脚本 `hadoop-daemon.sh` 中的 `rsync` 命令行是这样的:

```
rsync -a -e ssh --delete --exclude = .svn --exclude = 'logs/*' \  
--exclude = 'contrib/hod/logs/*' $HADOOP_MASTER/$HADOOP_PREFIX"
```

这里的可选项“`-e ssh`”表示与(`rsync` 的)对方即文件来源连接时采用 `ssh`,这里的文件来源可以就是我们这个主节点,也可以是个文件服务器。可选项“`-a`”就是“`-archive`”,表示整个目录的递归复制,也可以说是“全复制”的意思。另一个选项“`--delete`”,表示如果本地已有同名的文件(或目录)存在,就先加以删除。此外还有几个“`--exclude`”选项,是说跨节点全复制时要把某些文件或目录排除在外不予复制,这里排除的是运行日志,那当然不用复制。

复制完了整个 Hadoop 系统,下面就是在对方节点上按“`--script "hdfs"`”所说启动执行脚本 `hdfs`,后面还有参数“`start namenode`”,`hadoop-daemon.sh` 中的命令行是这样:

```
nohup nice -n $HADOOP_NICENESS $hadoopScript --config $HADOOP_CONF_DIR \
    $command "$@" > "$log" 2>&1 < /dev/null &
```

这里表面上好像看不到对脚本 `hdfs` 的启动, 其实 `$hadoopScript`, 即变量 `hadoopScript` 的值, 就是脚本 `hdfs` 的路径, 而变量 `command` 的值, 则是“start namenode”。至于 `nohup` 和 `nice`, 倒是 Linux 的 shell 命令, 前者表示所启动的进程不要一出错就挂断(hup)而退出运行, 后者表示把这个进程的优先级调降到变量 `HADOOP_NICENESS` 所述的级别。运行时不要挂断, 是有后面的重定向措施相配合的。这个进程的标准输出通道 `stdout` 被重定向到一个日志文件, 文件名就是变量 `log` 的值, 即“\$log”; 并且标准出错信息通道 `stderr` 即这里的通道 2 也被重定向到通道 1 即 `stdout`。这样, 在这个进程的运行过程中, 不管是提示信息还是出错信息, 就都会写入日志文件, 而不是在屏幕上显示。相应地, 从标准输入通道 `stdin` 的读入, 则被重定向到 `/dev/null`, 而不是等待键盘输入。最后, 这个命令行的末尾有个 `&`, 表示异步执行, 命令行一经启动就立即回到 Shell, 父进程不等待子进程结束就继续往下执行。换言之, 这个子进程一经启动, 就跟终端设备脱钩了, 从而也跟 `ssh` 实质上脱钩了。注意这是在目标节点上运行, 是目标节点上的脚本 `hdfs` 被启动了。

所以, 这个 `nohup` 的命令行实质上相当于“`hdfs start namenode`”。而脚本 `hdfs` 对于“start namenode”的响应, 如前一章中的摘要所列, 其核心的操作是“`exec "$JAVA" namenode`”。这里具体要运行的软件取决于变量 `JAVA` 的值, 一般就是 Java 虚拟机 `java`。之所以要有这么个变量, 是因为根据具体的配置有可能要采用其他版本或来源的名称不同的 Java 虚拟机, 对于 `DataNode` 还有可能采用安全性更高的 `jsvc`。

如果采用普通的 Java 虚拟机, 那么脚本 `hdfs` 中具体执行目标程序的命令行是这样:

```
exec "$JAVA"-Dproc_$COMMAND $JAVA_HEAP_MAX $HADOOP_OPTS $CLASS "$@"
```

这里的 `$COMMAND` 就是“namenode”, 与前缀 `proc_` 合在一起就是“`proc_namenode`”, 这将是所启动进程的名称。而 Java 虚拟机所实际装载运行的类, 则取决于变量 `CLASS`, 对于 `namenode`, 这个变量被设置成“`org.apache.hadoop.hdfs.server.namenode.NameNode`”。我们知道, `NameNode` 这个类带有 `main()` 函数, 是可以独立作为进程运行的。

注意, 这里 `ssh` 与 `rsync` 在 `slaves` 和 `hadoop-daemon` 这两个脚本中的划分, 脚本 `slaves` 中只对每个目标节点做 `ssh`, 让你登录到目标节点并在那里启动一个脚本, 那就是 `hadoop-daemon`, 至于这个脚本中做些什么事, 那与 `slaves` 无关。而 `rsync`, 还有对命令行“`hdfs start namenode`”之类的执行, 就都是在 `hadoop-daemon` 中。还要注意, `hadoop-daemon` 中对于 `rsync` 的执行是有条件的, 脚本中的相关代码是这样的:

```
if [ "$HADOOP_MASTER" != "" ]; then
    echo rsync from $HADOOP_MASTER
    rsync -a -e ssh --delete --exclude = .svn --exclude = 'logs/*'
        --exclude = 'contrib/hod/logs/*' $HADOOP_MASTER/ "$HADOOP_PREFIX"
fi
```

可见, 只是在环境变量 `HADOOP_MASTER` 给出了文件来源的时候才会执行 `rsync`, 如果没有这个环境变量, 或者变量的值是空, 那就不会执行 `rsync` 了。之所以这样安排, 是因为软件的部署只需一次, 而系统的启动运行却是比较经常的事。

而命令行如“hdfs start namenode”之类,则是在 start -dfs 中作为参数传下去的。脚本 hadoop-daemon 本身是中性的,你可以让它干这,也可以让它干那。而脚本 hadoop-daemons,则又把 slaves 和 hadoop-daemon 这二者综合在一起。

至于通过 hdfs 脚本所执行的命令是“start namenode”还是“start datanode”,那也只是对于 hdfs 脚本的不同参数而已,脚本只是因此而将变量 CLASS 设置成不同的值。

不过“stop namenode”和“stop datanode”有些特殊,这两种操作无须借助 hdfs 这个脚本,在 hadoop-daemon.sh 这一层就解决了。这是因为,要终止 namenode 或 datanode 的运行,只需把进程杀掉就可以了。但是怎么知道应该杀哪一个进程呢?原来,在脚本 hadoop-daemon.sh 中,在启动异步执行那个 nohup 命令之后(不等于进程结束就返回),紧接着还有个命令行“echo \$!>\$pid”,就是将所创建子进程的进程号保存在一个 pid 文件中。注意,这里的“pid”是个变量,它的值是个文件路径名。这样,只要检查一下,有这么个文件存在,其内容就是 namenode 或 datanode 进程的进程号(更确切地说是运行着 namenode 或 datanode 的 Java 虚拟机的进程号)。有了进程号,就可以 kill 这个进程了。事实上,hadoop-daemon.sh 中相关的代码是这样的:

```
if [ -f $pid ]; then          # 如果 pid 文件存在,节点上有 namenode 或 datanode 进程
    TARGET_PID='cat $pid'      # 从 pid 文件中获取进程号
    if kill -0 $TARGET_PID > /dev/null 2>&1; then # 先用“kill -0”试试
        echo stopping $command
        kill $TARGET_PID        # 先温和一点,kill 一下试试
        sleep $HADOOP_STOP_TIMEOUT # 小睡一会儿
        if kill -0 $TARGET_PID > /dev/null 2>&1; then # 如果进程还在
            echo "$command did not stop gracefully after
                $HADOOP_STOP_TIMEOUT seconds: killing with kill -9"
            kill -9 $TARGET_PID # “kill -9”是杀手锏
        fi
    else
        echo no $command to stop # 虽有 pid 文件,但实际上这个进程已经不存在
    fi
else
    echo no $command to stop # 无 pid 文件存在,节点上没有 namenode 或 datanode 进程
fi
```

明白了部署和启动 namenode 的过程,datanode 的部署和启动就不是个问题了。在前述的过程中,Hadoop 软件的部署主要是由 slaves.sh 和 hadoop-daemon.sh 完成的,而 namenode 和 datanode 的启动则由 hdfs 完成。其实脚本 hdfs 的作用远不止于 namenode 和 datanode 的启动,我们在上一章中看到,这个脚本的使用范围涵盖了所有的 HDFS 命令行操作。

再看 YARN 的部署和启动。虽然原理是一样的,但 YARN 的部署与启动比之 HDFS 还是有所不同。从大处着眼,YARN 的部署和启动主要也是分成两步,第一步是对 resourcemanager,第二步是对 nodemanager。但是不像 namenode 那样可以有好几个,这里假定 resourcemanager 只有一个,而且就是管理员所在的那个节点,所以用于 resourcemanager

部署和启动的脚本是只针对一个节点,并且是在当地运行的 yarn-daemon,而不是针对多个节点的 yarn-daemons。但是第二步针对 nodemanager 时的脚本就是 yarn-daemons。此外, start-yarn.sh 中还有个备用的第三步,是对 proxyserver 的部署和启动,也是用的 yarn-daemon,说明也是放在 resourcemanager 所在的节点上。

下面是 start-yarn.sh 的操作摘要:

```
yarn-project/yarn/bin/start-yarn.sh
> yarn-project/yarn/bin/yarn-daemon.sh start resourcemanager # on local node
>> rsync -a -e ssh --delete --exclude = .svn --exclude = 'logs/*'
    --exclude = 'contrib/hod/logs/*' $YARN_MASTER/ "$HADOOP_YARN_HOME"
    # 在本节点上执行 rsync,从$YARN_MASTER 拷贝
>>> cd "$HADOOP_YARN_HOME" # cd 到拷贝过来的目录中
>>> yarn-project/yarn/bin/yarn resourcemanager & # 启动脚本 yarn
>>>> exec "$JAVA" org.apache.hadoop.yarn.server.resourcemanager.ResourceManager
>>> echo $!>$pid # 将子进程的进程号记入 pid 文件, RM 已经启动
> yarn-project/yarn/bin/yarn-daemons.sh start nodemanager
>>> yarn-project/yarn/bin/slaves.sh ... yarn-daemon.sh ...
>>>> for each slave in `cat "$HOSTLIST"; do # 对于每一个 nodemanager 节点:
>>>>+ ssh yarn-daemon.sh start nodemanager # 远程登录,并要求在对方节点上执行脚本
>>>>+ yarn-project/yarn/bin/yarn-daemon.sh start nodemanager # 在远程节点上执行脚本
>>>>+ rsync
>>>>+ cd "$HADOOP_YARN_HOME"
>>>>+ yarn-project/yarn/bin/yarn nodemanager
>>>>+>> exec "$JAVA" org.apache.hadoop.yarn.server.nodemanager.NodeManager
>>>> done
> # yarn-project/yarn/bin/yarn-daemon.sh start proxyserver # 部署并启动 proxyserver
    //注意,在 Hadoop 源码的脚本 start-yarn.sh 中这一行是被注释掉的
    //如果需要可以去掉前面的 # 号,但要对配置文件做相应修改
>>> rsync -a -e ssh --delete --exclude = .svn --exclude = 'logs/*'
    --exclude = 'contrib/hod/logs/*' $YARN_MASTER/ "$HADOOP_YARN_HOME"
    # 在本节点上执行 rsync,从$YARN_MASTER 拷贝
>>> cd "$HADOOP_YARN_HOME" # cd 到拷贝过来的目录中
>>> yarn-project/yarn/bin/yarn proxyserver & # 用启动脚本 yarn 启动 WebAppProxyServer
>>>> exec "$JAVA" org.apache.hadoop.yarn.server.webproxy.WebAppProxyServer
>>> echo $!>$pid # 将子进程的进程号记入 pid 文件
```

先看 resourcemanager 的部署和启动。由于只对一个节点部署,并且是在本地,所以这里直接就执行脚本 yarn-daemon.sh,而不是 yarn-daemons.sh。所进行的操作也是 rsync,所用的选择项和参数也与前述基本相同。命令行中的参数 \$YARN_MASTER 是个“host: path”形式的路径,表示待部署软件的来源,rsync 就从这个节点拷贝软件至目标节点上。完成复制之后立即就运行脚本 yarn,使其启动 resourcemanager。

注意,这里的“`yarn-project/yarn/bin/yarn resourcemanager &`”只是摘要和示意,实际的命令行是这样的:

```
nohup nice -n $YARN_NICENESS "$HADOOP_YARN_HOME"/bin/yarn
--config $YARN_CONF_DIR $command "$@" > "$log" 2>&1 < /dev/null &
```

这跟前面启动 namenode 和 datanode 时所用的命令行几乎一模一样。这个命令行中变量 `command` 的值为 `resourcemanager`,这是从脚本 `yarn-daemon.sh` 传下来的;“`$@`”则为从上一层命令行传下的其余可选项和参数。

明白了 `resourcemanager` 节点的部署和启动,再看 `nodemanager` 节点,参考前面 `datanode` 的部署和启动,就容易理解了。同样,这里的脚本 `slaves.sh` 启动一个 `for` 循环,根据一个主机清单,对于用作 `nodemanager` 的节点逐一通过 `ssh` 跨节点登录并使其执行 `yarn-daemon.sh`。而脚本 `yarn-daemon.sh`,则在目标节点上执行 `rsync`,从指定的源节点拷贝以完成 Hadoop 的部署;然后以“`start nodemanager`”为命令行参数执行脚本 `yarn`,启动该节点上 `nodemanager` 的运行。注意,此时的脚本 `slaves.sh` 与前面部署 `hdfs` 时的所用并非同一文件,两个 `slaves.sh` 在不同的目录中,略有不同。另外,我们也可看出,此时的脚本 `yarn` 与前面所用的脚本 `hdfs` 处于同一层次并互相对应;二者都是在目标节点上运行,各自启动 HDFS 和 YARN 子系统。

上面说的是 Hadoop 的部署和启动,但是显然部署和启动是两回事,并非每次启动之前都得再部署一次,一般我们都只是单纯的启动,而且常常也不是整个集群的启动,而需要在个别特定的节点上启动。在这样的情况下,我们只需要通过 `ssh` 连到目标节点上,在目标节点上执行一次 `hdfs` 脚本和一次 `yarn` 脚本就行了。当然,在执行 `hdfs` 脚本时要根据该节点的角色给定命令行参数“`start namenode`”或“`start datanode`”。同样,在执行 `yarn` 脚本时则要给定命令行参数“`start resourcemanager`”或“`start nodemanager`”。

在此过程中基本上只用到两个脚本,分别用于两个子系统的运行,在 Hadoop 源代码中的文件路径是 `hdfs-project/hadoop-hdfs/src/main/bin/hdfs` 和 `yarn-project/yarn/bin/yarn`。事实上下面我们会看到,不仅仅是启动,对这两个子系统的日常使用也是依靠这两个脚本。

当然,也可以使用脚本 `slaves` 和 `hdfs-daemon` 加以成批启动。如前所述,差别只是要不要执行 `rsync`,而执不执行 `rsync` 其实取决于环境变量 `HADOOP_MASTER`。

19.3 Hadoop 的日常使用

软件系统的部署毕竟是较少发生的,而单纯的启动当然多一些,但是像 Hadoop 这样的系统常常是连续运行的,一经启动就会连续运行很长时间,所以更频繁发生的只是一般的日常使用。Hadoop 的源代码中为此提供了三个脚本:

```
common-project/hadoop-common/src/main/bin/hadoop
hdfs-project/hadoop-hdfs/src/main/bin/hdfs
yarn-project/yarn/bin/yarn
```

其中 `hdfs` 和 `yarn` 分别用于 HDFS 和 YARN 两个子系统的操作,脚本 `hadoop` 则比较宏观,相比之下前面两个用得更多。事实上前面完成部署之后的启动运行也是用了这两个脚本。

这三个脚本都是“就地”运行的,在哪个节点上运行,就是针对那个节点上的系统。不过就

地运行却不一定非得就地发起,使用者完全可以在另一个节点上通过 ssh 连到目标节点,发起这些脚本在目标节点上的运行。进一步,由于相应 Web 服务及其界面的存在,使用者还可以在别的节点上通过浏览器跨节点发起这些脚本在目标节点上的运行。

所以,这三个脚本,尤其是 hdfs 和 yarn,是使用十分频繁也很重要的脚本,我们有必要加以了解。

我们在上一章中已经摘要看过 hdfs 这个脚本,另一个脚本 yarn 的结构也基本相同,只是具体内容不同而已,其 Usage 提示是这样:

```
function print_usage(){
    echo "Usage: yarn [ -- config confdir] COMMAND"
    echo "where COMMAND is one of:"
    echo "    resourcemanager      run the ResourceManager"
    echo "    nodemanager           run a nodemanager on each slave"
    echo "    timelineserver        run the timeline server"
    echo "    rmadmin               admin tools"
    echo "    version               print the version"
    echo "    jar <jar>             run a jar file"
    echo "    application           prints application(s) report/kill application"
    echo "    applicationattempt    prints applicationattempt(s) report"
    echo "    container             prints container(s) report"
    echo "    node                  prints node report(s)"
    echo "    logs                  dump container logs"
    echo "    classpath             prints the class path needed to get the"
    echo "                          Hadoop jar and the required libraries"
    echo "    daemonlog             get/set the log level for each daemon"
    echo "    or "
    echo "    CLASSNAME             run the class named CLASSNAME"
    echo "Most commands print help when invoked w/o parameters."
}
```

对此无须再加解释。同样,如果 yarn 命令行中的 COMMAND 与脚本所提供的操作都对不上,那就是用户的自定义操作,由用户自己提供相应的 Java 类。实际使用中对 HDFS 子系统使用自定义操作倒不多,可是对 YARN 子系统就很多了,用户自己的 MapReduce 作业,包括 Hadoop 所提供的那些例子,对于 YARN 子系统都属于自定义操作。

至于脚本 hadoop 的 Usage,则是这样:

```
function print_usage(){
    echo "Usage: hadoop [ -- config confdir] COMMAND"
    echo "    where COMMAND is one of:"
    echo "    fs                    run a generic filesystem user client"
    echo "    version              print the version"
    echo "    jar <jar>           run a jar file"
```

```

echo "  checknative [-a|-h] check native hadoop and compression
                                libraries availability"
echo "  distcp <srcurl> <desturl> copy file or directories recursively"
echo "  archive - archiveName NAME - p <parent path> <src> * <dest>
                                create a hadoop archive"
echo "  classpath                  prints the class path needed to get the
echo "                          Hadoop jar and the required libraries"
echo "  daemonlog                  get/set the log level for each daemon"
echo " or "
echo "  CLASSNAME                  run the class named CLASSNAME"
echo ""
echo "Most commands print help when invoked w/o parameters."
}

```

这里有个 `fs` 命令,相应的 Java 类是 `org.apache.hadoop.fs.FsShell`,与前面 `hdfs` 脚本中的命令 `dfs` 相同。所以,命令行“`hadoop fs...`”与“`hdfs fs...`”是等价的。另外,在 `hadoop` 命令行中使用例如 `namenode`,那也是可以的,因为脚本 `hadoop` 会把这样的命令行转嫁给脚本 `hdfs`,下面是脚本 `hadoop` 中的有关片段:

```

COMMAND = $1
case $COMMAND in
...
namenode|secondarynamenode|datanode|dfs|dfsadmin|fsck|balancer|
fetchdt|oiv|dfsgroups|portmap|nfs3)
    echo "DEPRECATED: Use of this script to
                                execute hdfs command is deprecated." 1>&2
    echo "Instead use the hdfs command for it." 1>&2
    echo "" 1>&2
    # try to locate hdfs and if present, delegate to it.
    shift
    if [-f "${HADOOP_HDFS_HOME}"/bin/hdfs ]; then
        exec "${HADOOP_HDFS_HOME}"/bin/hdfs ${COMMAND}/dfsgroups/groups "$@"
    elif [-f "${HADOOP_PREFIX}"/bin/hdfs ]; then
        exec "${HADOOP_PREFIX}"/bin/hdfs ${COMMAND}/dfsgroups/groups "$@"
    else
        echo "HADOOP_HDFS_HOME not found!"
        exit 1
    fi
;;

```

可见,如果 `hadoop` 命令行中的命令是例如 `namenode`,并且脚本 `hdfs` 存在,脚本 `hadoop` 就会启动一个 `exec` 命令行,让其执行脚本 `hdfs`,将命令 `namenode` 转嫁给脚本 `hdfs`。

同样,对于某些针对 MapReduce 的操作,如果用户试图通过 `hadoop` 命令行启动,则 `hadoop` 会将其转嫁给另一个脚本 `mapred`。这个 `mapred` 脚本的 `usage` 是这样:

```
function print_usage(){
    echo "Usage: mapred [ -- config confdir] COMMAND"
    echo "      where COMMAND is one of:"
    echo "    pipes                run a Pipes job"
    echo "    job                  manipulate MapReduce jobs"
    echo "    queue                get information regarding JobQueues"
    echo "    classpath            prints the class path needed for running"
    echo "                        mapreduce subcommands"
    echo "    historyserver        run job history servers as a standalone daemon"
    echo "    distcp <srcurl> <desturl> copy file or directories recursively"
    echo "    archive - archiveName NAME - p <parent path> <src> * <dest>
                        create a hadoop archive"
    echo "    hsadmin              job history server admin interface"
    echo ""
    echo "Most commands print help when invoked w/o parameters."
}
```

最后,下面是两个命令行实例。

第一个是启动 `examples` 中计算圆周率 `pi` 的示例。其中的环境变量 `YARN_EXAMPLES` 必须事先已经设置好,要不然就得输入这个目录的全路径,那往往是很麻烦的。之所以要用 `jar` 而不是直接打入 `Java` 类名,是因为那些实例已经编译打包在一个 `jar` 文件中。第二个命令行则只是用来列出这个 `jar` 文件的内容:

```
~/bin/yarn jar $YARN_EXAMPLES/hadoop-mapreduce-examples-2.2.0.jar pi 16 1000
~/bin/yarn jar $YARN_EXAMPLES/hadoop-mapreduce-examples-2.2.0.jar
```

19.4 Hadoop 平台的关闭

最后是节点的关闭,如前所说,节点的关闭无须深入到脚本的 `hdfs` 或 `Yarn` 这一层,只要把 `nodenanager`、`namenode` 这些 `JVM` 进程 `kill` 掉就行。在启动这些进程时会留下 `pid` 文件,里面记载着进程号。或者,如果是人工操作,用 `ps` 命令看一下也不难。所以,如果是在个别节点上要关闭这个节点,是很简单的事。但是确实我们也要有一次操作就能关闭整个 `Hadoop` 平台的手段,`Hadoop` 为此提供了 `stop-all`、`stop-dfs`、`stop-yarn` 三个脚本。

我们从 `stop-all` 开始:

```
common-project/hadoop-common/src/main/bin/stop-all.sh
> hdfs-project/hadoop-hdfs/src/main/bin/stop-dfs.sh --config $HADOOP_CONF_DIR
>> hadoop-daemons.sh --hostnames "$NAMENODES" --script "$bin/hdfs" stop namenode
>>> common-project/hadoop-common/src/main/bin/slaves.sh
```

```

>>>> hadoop-daemon.sh
>>>>> TARGET_PID='cat $pid'; kill -0 $TARGET_PID //从 pid 文件中获得进程号后杀掉
>> hadoop-daemons.sh --hostnames "$NAMENODES" --script "$bin/hdfs" stop datanode
>>> common-project/hadoop-common/src/main/bin/slaves.sh
>>>> hadoop-daemon.sh
>>>>> TARGET_PID='cat $pid'; kill -0 $TARGET_PID
>> hadoop-daemons.sh --hostnames "$SECONDARY_NAMENODES" \
--script "$bin/hdfs" stop secondarynamenode
>>> common-project/hadoop-common/src/main/bin/slaves.sh
>>>> hadoop-daemon.sh
>>>>> TARGET_PID='cat $pid'; kill -0 $TARGET_PID
>> hadoop-daemons.sh --hostnames "$JOURNAL_NODES" \
--script "$bin/hdfs" stop journalnode
>>> common-project/hadoop-common/src/main/bin/slaves.sh
>>>> hadoop-daemon.sh
>>>>> TARGET_PID='cat $pid'; kill -0 $TARGET_PID
>> hadoop-daemons.sh --hostnames "$NAMENODES" --script "$bin/hdfs" stop zkfc
>>> common-project/hadoop-common/src/main/bin/slaves.sh
>>>> hadoop-daemon.sh
>>>>> TARGET_PID='cat $pid'; kill -0 $TARGET_PID
> yarn-project/yarn/bin/stop-yarn.sh --config $HADOOP_CONF_DIR
>> yarn-daemon.sh --config $YARN_CONF_DIR stop resourcemanager
>>> TARGET_PID='cat $pid'; kill -0 $TARGET_PID
>> yarn-daemons.sh --config $YARN_CONF_DIR stop nodemanager
>>> common-project/hadoop-common/src/main/bin/slaves.sh
>>>> yarn-daemon.sh
>>>>> TARGET_PID='cat $pid'; kill -0 $TARGET_PID
>> yarn-daemon.sh --config $YARN_CONF_DIR stop proxyserver
>>> TARGET_PID='cat $pid'; kill -0 $TARGET_PID

```

我们已经知道 `hadoop-daemons`、`slaves` 和 `hadoop-daemon` 这些脚本的作用和内情，也知道像 `namenode` 一类进程的进程号记录在 `pid` 文件中，这里就不用多说了。不过，从逻辑上说，`stop-yarn.sh` 似乎应该放在 `stop-dfs.sh` 之前。

第20章

Spark 的优化与改进

20.1 Spark 与 Hadoop

Hadoop 问世以后,很快便引起了人们的注意并得到采用。这一来是因为当时确实有了大数据处理的需求,尤其是对于大量非结构数据进行处理的需求;二来也因为 Hadoop 本身是个质量比较高的开源项目,其结构和功能与 Google 论文中描述的系统很接近,而后者却既不开源也非商品。此外,当时如 *Hadoop the Definitive Guide* 等书籍的出版也对 Hadoop 的普及起了重要的作用。

但是人们也很快就发现,对于许多应用而言 Hadoop 存在着诸多不足,于是就有人试图加以优化和改进,这方面最突出的成果当数 Spark。另一方面,Hadoop 自身也在不断提高和改进,其 YARN 框架和 Streaming(即流处理)模式就是这样来的。近年来,Hadoop 与 Spark 之间就像展开了一场你追我赶的竞赛。时至今日,Hadoop 和 Spark 已经成为大数据处理平台的两个“de facto standard”,即“事实标准”。不过 Spark 之于 Hadoop 也并非完全是对立的两种平台或产品,在很大程度上倒是对于 Hadoop 的补充,而并不完全是作为对于 Hadoop 的替代。事实上,Spark 虽然也能以“Stand alone”模式独立存在和运行,但是更多地还是利用 YARN,在 YARN 框架上运行。而且 Spark 也不提供自己的文件系统,大多只是直接利用 HDFS。虽然 Spark 并不要求必须使用 HDFS,但是在大规模集群的条件下要实现“数据在哪里,计算就去哪里”这个原则,而且还要容错,实际上也没有太多的选择。所以从功能上看,Spark 的作用只是相当于一个更好的 YARN 子系统。

Hadoop 的不足是明摆着的,总而言之,一是不够灵活、比较死板,就是专门针对 MapReduce;二是性能不够好;三是使用不够方便,动不动就得写个 Java 程序。

那么 Spark 对此又有些什么样的改进呢?下面就作些介绍和评述,同时也对 Hadoop 和 Spark 做个粗泛的比较研究。

20.2 RDD 与 Stage——概念与思路

与 Hadoop 相比,Spark 在关于数据流的观念与思路方面有了创新和改进。在传统的数据流结构中,虽然称为“数据”流,但实际上人们的眼睛还是盯着对于数据的操作和运算,而非数据本身。仍以前面所举的假想 Unix/Linux 管道线“cat file | normalize | sort | uniq”为例,这是个一维的链状数据流,但是在这里我们关注的重点仍是每个节点对于所流经数据的处

理,即每个节点对其输入数据做了何种操作和运算,而并非数据本身。在传统的数据流图中,对数据的操作和运算以一小圈表示,是图中的“顶点(Vertex)”;而先后两步操作之间的数据传输,则只是一条连线,即“边(Edge)”,表示这中间有数据的传输和流通。虽然并没有说这二者孰轻孰重,但一般都把注意力更多地放在点上。至于数据本身,则根本就是无形的。这样,所谓“数据”流,实际上倒是“操作”流。在 Unix/Linux 的管道线中,字符“|”代表着两步操作之间的数据流动,代表着前后两个进程间的 IPC,实际上也代表着作为中间结果的数据,但却是隐匿的而无法加以引用。前面讲过,Unix/Linux 管道线之所以只能是链状的,而不能分叉、不能是 DAG,这就是最重要的原因。在跨机器节点并行处理,并且前后两级的计算节点之间是多对多(例如 M 个 Mapper 和 N 个 Reducer)的环境和系统中,这个问题就显得尤为突出,此时在两个(概念上的)节点间,实际上是两个计算步骤之间,流通的不是单项数据,而是数据的集合(设想 M 个 Mapper 并行的输出)。

那么,从系统结构的角度,我们是否应该反过来把注意力更多地放在数据上,真正形成“数据”流的概念及其实现呢?就如“对象(Object)”是“数据结构与定义于其上的操作”的概念一样,我们也应该有个“数据的集合与定义于其上的操作”。这样,我们在考虑系统结构的时候,更应该关注的是:一个数据集合如何因为计算处理而转变成另一个数据集合,这另一个数据集合又如何流转到从事下一步计算的机器节点(集合)上,在那里又经计算处理而转变成另一个数据集合。通过分析 Spark 的源码,我们可以推断它的设计思路应该就是这样来的。

在 Spark 中,这样的数据集合称为 RDD,就是“Resilient Distributed Datasets”,即弹性分布数据集。这里的 Resilient 一词意为“有弹性能复原”,也有“不抱怨”的意思;Distributed 则是说具体数据集合中的数据分布在多个机器节点上,它们的总和就是这个数据集合。以我们熟知的 MapReduce 为例,假定有 M 个 Mapper 和 N 个 Reducer,这 M 个 Mapper 的输出数据的集合就是一个 RDD;但是每一个 Mapper 的输出数据都要按某种规则分发到 N 个 Reducer 上去,这称为 N 个“分区(Partition)”,所以每个 Mapper 的输出数据都要因分区而先被置入 N 个假想的篮子(Bucket)。这样一共就有 $M \times N$ 个 Bucket,这些篮子中的数据将被分发到 N 个后续计算节点,例如 Reducer 节点。而每个后续计算节点,则各自从 M 个前导计算节点把属于自己的那个篮子中的数据拷贝过来,这就是一个分区即 Partition 的数据,这个过程就是 Shuffle。如前所述,Shuffle 是个开销颇大,并且可能引入明显延迟的环节。经过 Shuffle 之后,仍以 MapReduce 为例, M 个 Mapper 节点的输出数据就到了 N 个 Reducer 节点上,也成为集合、一个 RDD。与 M 个 Mapper 输出端上的 RDD 相比,这个 RDD 的组成不同了,属于相同 Partition 的 M 个篮子中的数据都集中到了同一个 Reducer 节点上,但是它们的总和并没有变,实质上还是同一个 RDD。所以 Shuffle 只改变 RDD 的形态,而不改变这个集合的内容。

由于 Reducer 节点有 N 个,是个集合而并非单个,所以称其操作 `reduce()` 为数据流中的“一个”操作其实并不合适,应该是“一步”操作,这是数据流中的一个阶段、一个步骤。

再回头看看 M 个 Mapper 的输入,我们知道 Mapper 阶段的输入是分成片,即 Split 的,Split 与 Partition 其实并无本质的区别,所以 M 个 Mapper 的输入数据集合也是一个 RDD,只不过数据的分区取决于它们在输入文件中的位置,并且对此 RDD 所做的处理是 `map()` 而已。所以,Map 这个阶段所做的计算把一个 RDD(不妨称为“Map 前 RDD”)转化成了另一个 RDD(不妨称为“Map 后 RDD”)。经过 Shuffle 之后到了 Reduce 阶段的输入端,则又成为另一个实质相同但组成不同的 RDD(不妨称为“Shuffle 后 RDD”)。而 Reduce 阶段则对其输入 RDD 所

进行的处理则有所不同,并不是把它转化成另一个 RDD,而是将其转化成输出结果,这样的操作称为 Action。

但是如果我们细察 Mapper 这个阶段,则又可发现其实里面有更多的 RDD 转化,例如每个 Mapper 的输出虽然经过 Partitioner 的分区而进入 N 个篮子,却是未经排序的。经过排序之后,虽然还是那些数据,还是同样的分区,但已经又是一个不同的 RDD 了,所以这也是一次 RDD 的转化。同样的道理,Reduce 阶段的输入端有对于来自不同 Mapper 节点而又属于同一分区的数据的合并排序,经过合并排序之后又是一个不同的 RDD,这又是一次 RDD 转化。

这里讲的还只是从单个 RDD 到单个 RDD 的一对一转化,实际上还可能需要有二对一的转化,例如两个集合的相加、相减、相乘、合并等。有了两个集合之间的操作和运算,就可以实现例如把数据流中的两个分支聚合在一起。显然这些需求在 Hadoop 中都被忽略了。而且,在 Hadoop 中,像排序这样的转化对于应用层都是不可见的,或者可以说是强加给应用层程序员的,把这些功能和步骤都交给使用者,让他们能根据实际需要灵活搭配,显然很有意义。

再深入一些,我们还可发现,把一步 RDD 转化的输出(是个 RDD)变成下一个 RDD 转化的输入(也是个 RDD),这个过程有两种不同的情况。一种是前后之间 Partition 的组成有变化而需要经过 Shuffle 的,就像 Mapper 中经过排序之后的 RDD 与 Reducer 中合并排序之前的 RDD 之间那样。另一种是前后之间 Partition 的组成没有变化而无需经过 Shuffle 的,例如 Mapper 节点上 `map()` 函数的输出,即未经排序之前的 RDD,与作为排序阶段输入的 RDD 之间,就无需 Shuffle,并且事实上就是同一批数据结构。

作为一种特例,如果把两个 Mapper 即 Map1 和 Map2 串在一起,二者的节点数量相等,并且前后之间是一对一直连的关系,不需要有 Shuffle,那么 Map1 输出端的 RDD 就是 Map2 输入端的 RDD。

显然,前后之间无需经过 Shuffle 的两步 RDD 转化完全可以放在同一组机器节点上完成,就其中的一个特定的 Partition 而言就是在同一台机器上,就像 Hadoop 中的 `map()` 计算和排序就是在同一机器节点上。

这样,从一个 RDD 开始,一个节点承担其中一个 Partition 的计算,一步步转化,只要转化的结果是无需 Shuffle 的 RDD,Partition 的组成和形态没有变化,就可以安排在同一个机器节点上;直至产生一个需要 Shuffle 的 RDD,由于 Partition 的组成和形态有了变化,得要跨节点进行 Shuffle,就不能放在同一个节点上了。像这样可以被安排在同一个节点上的一段 RDD 序列,就是宏观意义上的一“步”计算,或者说一个计算阶段,在 Spark 中称为 Stage。

于是整个计算、整个数据流(或工作流)中的一个链状分支就是若干 Stage 的级联。例如 MapReduce,就是 Map 和 Reduce 的级联。这里 Reduce 阶段的输出并非 RDD,严格说来最后一步并非 RDD 转化而是 Action,但是从系统结构的角度我们也不妨将其视同 RDD 转化。这样,如果把这些要素都交到应用程序员的手里,他们就可以构建出例如 Map1Map2Reduce、Map1Map1Map1Reduce 那样的数据流(工作流)。另一方面,RDD 作为有命名、可引用的变量,就可以成为分叉点,从而为 DAG 的构建提供了条件,因为一个分支其实就是一个以此 RDD 为起点的 Stage 序列。而且,如果把某个 RDD 持久化存储起来,以后就可以重启以这个 RDD 为起点的计算。

以前我们常常把一连串微观的操作称为一个数据流,现在不妨以一连串宏观的 Stage 来代表一个数据流。而 Stage 的序列实际上是 RDD 的序列,而且单个 Stage 本身也是个 RDD

序列。这样的 RDD 序列称为 RDD Dependency, 即“RDD 依赖”, 因为序列中的每个具体 RDD 都依赖于它前面的那个 RDD, 都是在其前导 RDD 的基础上经过某种转化运算 (Transformation) 而得到的。而最终的计算结果, 则是以这个序列中最后一个 RDD 为基础经过某种动作运算 (Action) 所得。RDD 依赖可长可短, 但不可以成环。而系统的结构和框架, 则应能适应这样长短不一的 RDD 依赖序列的实现。

进一步, 如前所述, 由于 RDD 的存在是一种显式的存在, 可以作为变量加以引用, 这个数据流就可以分叉, 从某个 RDD 上分支出另一个 RDD 依赖序列。这样, 整个作业在宏观上就成了一个 RDD 依赖图, 那就是一个 DAG, 即有向无环图, 图的每个分支都是一段 RDD 依赖序列或者一个 RDD 依赖子图。

再进一步, 同一个 Stage 或同一串 Stage, 在数据流中还可重复出现, 这就为迭代 (Iterative) 计算提供了条件。这对于机器学习尤为重要, 许多机器学习的算法都需要迭代。显然, 比之 Hadoop, 这样就大大提高了系统在结构和应用上的灵活性。

由 Stage 级联构成的计算流程, 本质上仍是批处理, 仍是工作流而不是数据流; 因为不仅其中的每一小步都是针对整个 RDD 的操作, 而且前一个 Stage 的输出数据在其最后一个 (输出) RDD 中聚集了之后才会启动 Shuffle。

经过 Shuffle 之后, 前一阶段的使命即已完成, 这个阶段的计算所占的资源就可以释放出来了, 但是这个阶段所输出的 RDD 又会触发下一个阶段的计算。这样, 对于由多个 Stage 级联形成的计算流程, 就可以像饭馆里生意兴旺时要“翻桌”那样, 按时间顺序将新来的 Stage 部署在相同 (或不同) 的一组节点上。此时的并行性存在于并行从事同一 Stage 的计算甚至同一 RDD 转化的许多不同 Partition 之间, 也可能存在于同一 Stage 内部的前后不同的 RDD 转化之间, 但不存在于不同 Stage 之间, 因为前面的 RDD 转化不完成就不会启动后面的 RDD 转化。至于排序, 可以有也可以无, 反正本来就是批处理, 加上排序也只是增加了排序本身所需的运算量。

可以说, Spark 对于 Hadoop 的优化和改进, 根源都可以追溯到这个观念的转变, RDD 和 Stage 概念和机制的引入及实现带来了许多好处, 下面还要加以介绍和讨论。

20.3 RDD 的存储和引用

上面讲到, Spark 引入了 RDD 和 Stage 的概念, 将整个计算过程分解成若干个阶段或步骤, 即 Stage; 每一个 Stage 都有个输入端 RDD, 对这个 RDD 进行某种操作就将其转化成另一个 RDD, 又进行某种处理就又转化成另一个 RDD, 直至得到一个需要加以 Shuffle 的输出端 RDD。前后两个 Stage 之间是 RDD 的传输, 一般而言这是个 Shuffle 的过程。对计算过程中 Stage 的数量则并无限制。

每个 RDD, 实质上是分布于多个机器节点、逻辑上相关联的一组数据缓冲区, 其中的每个缓冲区就是这个 RDD 的一个 Partition。这组缓冲区对于用户以及在 App 程序中是可以有命名, 从而可以被引用的, 不过一旦生成之后便是只读, 就像一个 Immutable 的变量一样。

Spark 的设计目标是要让这样的数据集合常驻内存, 尽量避免写入文件, 以提高效率。事实上 Spark 就是以“内存计算”为标榜的, 意思就是其计算的中间结果都在内存中而无须写入磁盘文件。不过这并不意味着 Spark 就从来不把 RDD 写入文件。

首先,把中间结果写入磁盘往往是被迫的,因为内存的大小总是有限,容纳不下的时候只好把数据“溅出(spill)”到磁盘文件中,这跟操作系统把存储页面 swap 到磁盘上是一样的道理。所以,归根结底,问题在于是否在内存中累积起太多的数据。RDD 是数据的集合,数据跨节点传输的粒度是整个数据集合 RDD(对于下游个别的具体节点而言是一个 Partition),而不是单项数据。所以基于 RDD 的处理仍是批处理(Batch Processing)而不是流处理(Stream Processing),是工作流而不是数据流,那么多的数据堆积在内存中,就难免会有 spill 的需要。

与 Hadoop 相比,既然都是批处理,都要在内存中堆积起整个数据集合,那 Spark 就没有什么特别的优势可以减少 spill。程序上的优化或许可以在这方面有所改进,但难有实质性的提高。为解决这个问题,也为提高数据处理的实时性,就有必要减小数据跨机器节点传输的粒度,理想状态是减小到单项数据,变工作流为数据流。这样,数据在处理的过程中就能像工业流水线上的道道工序一样,材料一到就加工,加工了就走,那样(在理想状态下)就不会有堆积,自然也就没有 spill 的要求了。为此,与 RDD 相对应,Spark 又提供了基于 DStream 的计算模型,这就更接近于本来意义上的数据流模型了,对此下一节中还要加以介绍。

另外,我们有时候也有主动把 RDD 存入磁盘的要求。这一方面是出于引用作为中间结果的 RDD 的需要,在大数据处理中常常需要把一个作业的某一步中间结果持久化存储起来,供另一个作业作为输入,因为大数据处理,特别是数据挖掘的一个本质特征就是试探,这个办法不行就换一个办法,但是未必需要每次都从头开始。另一方面这也是出于容错的需要,把一个 RDD 持久存储起来,就成为整个计算流程中的一个 Checkpoint,如果后面的计算因故失败,就可以从这个 Checkpoint 处加以恢复。但是当然不必也不应把每个 RDD 都持久存储,因为那样代价太大(须知这是大数据),需要时宁可从某个作为 Checkpoint 的 RDD 开始重新加以计算。所以,RDD 这个类提供了一个方法 computeOrReadCheckpoint(),就是看一下这个 RDD 是否曾作为 Checkpoint 加以存储,如果是就 Read,要不然就回溯到它前面的某个 Checkpoint 开始重新加以计算。

20.4 DStream

如前所述,由 Stage 和 RDD 级联构成的计算流程,本质上仍是批处理,仍是工作流而不是数据流,因为 Shuffle 这个数据传输过程的粒度是整个 RDD。

为提高系统的实时性,并进一步提高系统的并行度,更为避免被动的 spill 操作,就有必要减小数据跨机器节点传输的粒度,理想的状态是减小到单项数据,彻底变工作流为数据流。这样,数据在处理的过程中就能随到随走,没有数据的堆积,既不占用内存,也把数据处理的延迟降到最低。为此,与 RDD 相对应,Spark 又提供了基于 DStream 的计算模型,这才像是本来意义上的数据流模型了。DStream 意为“Discretized Stream”,即离散化的 Stream,因为这毕竟还不是像水流那样彻底的连续,在理想状态下也只是一项项的离散数据,例如一个个的单词、一个个的 KV 对。一个 DStream 相当于一个动态的、其内容在“连续”流动的 RDD,最初可以来自文件的读出、网络通信 Socket 上的读入、实时生成的数据等连续的数据源。或者,也可以把它想像成按时间先后对数据流进行切片。

但是真要把跨节点传输数据的粒度降到单项数据也不现实,那样数据传输的开销就很大,我们总不能将每个单词都打成一个 IP 包发送。所以 Spark 采取了折中的方案,数据传输和处

理的粒度既非单项数据,也非全部数据,而是一个小批量接着一个小批量的数据,称为一个个的 batch。一个 batch 就相当于一个切片,每个 batch 的大小可以由用户和 App 的程序员自定,比方说确定为这个流中每 1 秒钟或 2 秒钟的流量。而这 1 秒钟或 2 秒钟流量中的数据,就成为一个 RDD。这样,一个数据流(Stream)就按时间分解成一个 RDD 的序列,如果这个 Stream 的流量永远都没个完,就永远有新的 RDD 被生成。注意,这个 RDD 序列与前述的 RDD 依赖序列是两码事,这个 RDD 序列中的 RDD 之间只有时间关系,而没有计算上的依赖关系。这样一来,Spark 对每个 RDD 的处理严格说来仍是批处理,但现在是许多容量不大的 RDD,是粒度较小的批处理。当然,每个这样的 RDD 仍被分成许多 Partition,由许多节点并行加以处理。另一方面,这样的 RDD 仍可根据具体的计算流程一步一步转化成别的 RDD,并按计算的先后和是否需要 Shuffle 而组合成 Stage,使对于流的计算转化成对 Stage 序列的计算。而 Spark 的调度机制,则依次部署这些 Stage 运行。在理想的情况下,可以在前一个 Stage 结束运行之后马上就调度下一个 Stage 运行,连地方都不用换。实际上则未必能那样天衣无缝,因而实际在运行的通常是一个“滑动窗口(Sliding Window)”,即连续几个 Stage,最老的那个一结束就换上最新的。

这使 Spark 的实时性有了较大的改进,因为虽然还没有达到纯粹意义上的数据流处理,但毕竟也不是完全意义上的批处理即工作流了。当然,这里的前提是不能有排序,有排序就一定会数据的堆积。

一般认为 Spark 的实时性比 Hadoop 好,可以用于有实时要求的应用,这是拿基于 DStream 的计算去跟 Hadoop 的 MapReduce 框架相比,后者实际上就是基于 RDD 的计算,而且还带全局排序。其实新版本的 Hadoop 也有 Stream,只是不如 Spark 那样健全。

20.5 拓扑的灵活性和多样性

RDD/Stage 概念的引入为 Spark 的计算带来了数据流拓扑灵活性方面的改进。一来是流的长度不再像 MapReduce 框架那样地固定和僵化,而可以动态地把多个 Stage 级联起来,流的长度动态可变了。而且,由于 RDD 可命名可引用,数据流就可以分叉形成有向无环图 DAG 了。进一步,多个 Stage 的级联并不排除 Stage 的重复级联,也不排除 Stage 序列的重复级联,因而就可用于迭代计算,这一点对于机器学习有着重要的意义。其实,如前所述,Spark 还允许两个分枝的汇聚。

20.5.1 数据流分叉与 DAG

RDD 这个概念的引入有个极大的好处,就是数据流可以分叉了。我们不妨假想 Unix 上的例如这样一个流水线:

```
cat file2 | sort | uniq | wc -l
```

这个流水线的运行结果让我们知道 file2 中出现了多少个不同的单词,包括单复数和时态的不同、大小写的不同,都算在内。但是如果我们同时也想知道实际上这里面有多少词汇,例如 do、did、done 其实是同一个词汇,那么我们或许可以写一个程序 normalize,将例如 did 和 done 都规范成 do。但是大小写没有必要与时态和单复数混在一起,那要简单得多,所以我们

又另写一个小程序 upper, 将所有字母一律换成大写。然后我们希望能建立起一个相当于这样的数据流拓扑:

```
cat file2 | upper | sort | uniq | wc -l
                | normalize | sort | uniq | wc -l
```

就是说, 共用其中的“cat file2 | upper”这一段, 然后分叉, 一个分支直接执行“sort | uniq | wc”, 另一个分支先做 normalize, 然后也是“sort | uniq | wc”。读者也许说, 何必呢, 分两次做不就行了吗? 共用“cat file2”就有那么重要? 对于单机上的小数据计算确实不重要, 但是对于大规模集群上的大数据文件(设想 file2 的大小可能有 1TB, 分布在数百个节点上), 这就不一样, 不能忽略不计了。再说, 这里还只是举一个简单的例子来说明问题, 实际需求常常远为复杂。

但是, Unix 的 Shell 显然不支持这样的分叉, 它的 Shell 语言中没有这样的语法和语义。那么能不能将 upper 的输出当作一个变量, 然后让后面的两个分支都来引用这个变量作为其输入? 这就走到 RDD 的思路上了, 但是 Unix 的 Shell 语言没有这样的语法和语义。在 Unix 上, 我们能做的是例如这样:

```
cat file2 | upper > file2-upper
cat file2-upper | sort | uniq | wc -l
cat file2-upper | normalize | sort | uniq | wc -l
```

这还是有点不一样, 一来这是强制要额外读写文件, 二来在集群条件下也使计算的分布变得更加复杂。所以, 让数据流可以分叉, 是 RDD 的一个重要作用。惟其如此, Spark 才可以说是支持 DAG, 即构成有向无环图的数据流。

如前所述, Spark 还支持对两个输入 RDD 进行某些计算, 将之转换成单个输出 RDD。这样就可以把两个数据流分支汇聚在一起, 这实际上已经超出了树状 DAG 的范畴。

20.5.2 MapReduce 计算的级联

早期的 Hadoop 只支持 MapReduce 一种计算模型, 即只支持长度为 2 的链状数据流, 这当然也是一种缺陷。不过后来的 Hadoop 已经有了改进, 也开始支持“流模式”, 实际上就是支持 Unix 的链状流水线, 长度不再局限于 2。但严格说来这还是有所不同, 因为 Hadoop 的流模式实际上是多个链状数据流的并联, 就好像是拿好几股绳子并成了一根, 这很多股分支之间是不能有 Shuffle, 甚至一般而言不能有跨节点数据传输的, 这跟 Spark 中多个 Stage 的级联还是不一样, 当然更不能说是支持 DAG。

既然 Hadoop 的主要计算模型是 MapReduce, 那么自然就会有级联(Cascading)相同或不同 MapReduce 计算的要求。Hadoop 的示例程序中有个 Grep, 就反映了这样的要求, 我们为它做个摘要:

```
class Grep extends Configured implements Tool {}
] run(String[] args)
> Job grepJob = new Job(conf)
> FileInputFormat.setInputPaths(grepJob, args[0])
// 整个 Grep 操作的数据输入文件路径来自命令行
> grepJob.setMapperClass(RegexMapper.class)
```



```

> grepJob.setCombinerClass(LongSumReducer.class)
> grepJob.setReducerClass(LongSumReducer.class)
> FileOutputFormat.setOutputPath(grepJob, tempDir)
// 第一个 MapReduce 计算的输出写入临时目录中的文件
> grepJob.setOutputFormatClass(SequenceFileOutputFormat.class)
> grepJob.setOutputKeyClass(Text.class)
> grepJob.setOutputValueClass(LongWritable.class)
> grepJob.waitForCompletion(true) // 等待第一个 MapReduce 计算 grepJob 的完成
> Job sortJob = new Job(conf) // 然后开始第二个 MapReduce 计算 sortJob
> sortJob.setJobName("grep - sort")
> FileInputFormat.setInputPaths(sortJob, tempDir)
// 第二个 MapReduce 计算的输入来自临时目录中的文件
> sortJob.setInputFormatClass(SequenceFileInputFormat.class)
> sortJob.setMapperClass(InverseMapper.class)
> sortJob.setNumReduceTasks(1); // write a single file
> FileOutputFormat.setOutputPath(sortJob, new Path(args[1]))
// 整个 Grep 操作的数据输出文件路径也来自命令行
> sortJob.waitForCompletion(true)

```

整个 Grep 的过程是两个 MapReduce 的级联。第一步是真正的过滤抽取,即 grepJob 所做的事,这是一个 MapReduce 计算。第二步是对第一步输出结果的排序,即 sortJob。这里调用 waitForCompletion() 两次,整个过程是作为两个作业,分两次提交的。宏观地看这仍是一个工作流,但是从系统结构的角度看就连这样的工作流也得在框架之外由 App 程序员操心,在 App 程序中加以变通才能实现。这使人不禁想起一幅图景,就是打一枪就得拉一下枪栓、上一粒子弹。为什么呢?就是因为 Hadoop 的 MapReduce 框架针对性太强而灵活性太差。

而在 Spark 的框架中,这就不是个问题。Spark 支持 DAG 数据流,而链状数据流只是其子集,对于链的长度也并不加以限制。链状数据流本来就是多个计算步骤的级联,当然也可以是 map1().reduce1().map2().reduce2() 或者 map1().map2().reduce() 这样的级联。

不过也要看到,同样的计算在 Hadoop 中并非不能实现,只是方便的程度不一样,这有点像高级语言与汇编语言的差别。

20.5.3 计算的迭代和循环

与 MapReduce 的级联相比,对 MapReduce 的迭代和循环也同样重要,也许更加重要。这是因为,特别是对于机器学习,迭代和循环都是很重要的手段,而迭代归根结底是一种循环。例如一种常会用到的 K-means 聚类算法,就需要有多轮迭代,这实质上就是个 while 循环,一轮一轮地计算直至满足条件为止,但是每一轮计算中所用的算法是一样的,不同的只是数据或参数。如果循环中的每一轮计算都是通过 MapReduce 完成的,则所用的 Mapper 和 Reducer 不变,即算法不变。如果是把上一轮循环的输出用作下一轮循环的输入,那就是迭代,要不然就是一般的循环。

显然,迭代计算的要求与 DAG 的拓扑架构是冲突的,因为既然是“无环”就不允许有反馈

的通道,而迭代恰恰要求将上一轮计算的输出反馈回去用作下一轮计算的输入。

Hadoop 的 MapReduce 框架自身显然不支持迭代,这当然是个缺陷。要解决这个问题,可以有两种方案,一种是仍保持 MapReduce 的拓扑架构不变,但是类似于上述的示例 Grep 中那样,把前一轮 MapReduce 计算的结果写入文件,再将这个文件用作下一轮 MapReduce 的输入,但是得把这个过程隐藏在 API 后面,不要让 App 程序员来操心如何实现这样的迭代。从本质上说,这只是在使用界面和语言表达上解决,让使用者感觉到 MapReduce 这个框架也可以迭代。另一种则是彻底的解决,从计算框架上实现输出的反馈,变有向无环为有向有环。当然,后者的设计与实现要复杂得多。所以人们都倾向于采用前面这种简单的方案。曾经有个开源项目叫 HaLoop(见 sigmod.github.io/papers/HaLoop_camera_ready.pdf),显然就是“具有 Loop 功能的 Hadoop”的意思,所采用的就是这样的方案。

至于 Spark,由于对 Stage 的级联更为灵活和方便,自然可以通过相同 Stage 的级联来实现迭代,但是本质上与 HaLoop 采用的方案没有什么不同,只是程序上和表达上显得更方便更灵活,也更自然。当然,Spark 所做的优化和改进远不是 HaLoop 所能比拟。

20.6 性能的提升

前面讲述 Hadoop 的数据流那一章中讲到数据在 Mapper 与 Reducer 之间的那一段流程:Mapper 的输出会被排序后写入 spill 文件,如果 Reducer 不止一个,即有多个 Partition 就会有多个 spill 文件,到所有(可能有很多)Mapper 各自把所有的输出都写入并且合并成一个总的 spill 文件之后,具体的 Reducer 这一边就会通过 Shuffle 把各个 Mapper 针对该具体 Partition 的 spill 文件内容都拷贝过来,并加以合并排序,然后作为这个 Reducer 的输入。在 Hadoop 看来,把 Mapper 的输出写入 spill 文件似乎是理所当然的,因为既然是“大数据”,那自然很难在内存中容纳下来。当然 Hadoop 也不傻,它也要判定一下,内存配额中容纳不下才不得已而写到 spill 文件中,而且内存配额的大小可以根据具体情况加以设置。

毫无疑问,写不写文件对于性能有着重大的影响。而 Spark 强调和标榜的就是“内存计算”,即尽量避免将中间结果写入文件。

然而,对于同一个输入数据集合、同样大小的内存,如果在 MapReduce 框架中因为内存中容纳不下而不得不“溅出(spill)”到磁盘文件中,那么在 Spark 的框架中怎么就可以不用写文件了呢?如上所述,这是靠 DStream 才得以实现的。

在 Hadoop 的数据流(工作流)中,排序是个核心的症结。一方面,排序的要求使 Map 和 Reduce 这两个阶段只能是批处理,只能是工作流而不是数据流,所以事实上 MapReduce 的框架根本就不是数据流的框架。而既然是批处理,那就得在两个阶段之间积累和缓存整个数据集合,这就使得内存成为瓶颈。另一方面,排序本身就是不小的开销,对于大数据的数据量,这个负担显然不小。而且,在 Hadoop 中,排序操作在很大程度上是强加的,是不让用户选择的,因为事实上用户很少有可能把 MapOutputBuffer 和 Shuffle 这两个类都用不排序的设计给替换掉。

相比之下,在 Spark 中,不管排序不排序,在 Shuffle 之前终归也要堆积起整个 RDD,终归也是批处理,不排序只是减少了排序本身的开销,而并不会因此就不需要有 spill 而得以避免写入文件。所以,说是因为有了 RDD 就可实现“内存计算”,从而大大提升性能,是并不令人

信服的。但是采用了 DStream,情况确实就不一样了。

DStream 按给定的时间长度把输入流切割成一连串小块的 RDD,再依次对这些 RDD 进行计算。虽然对具体 RDD 的处理仍是批处理性质,但是现在的 RDD 小了,可以来了就处理,处理完就走,在内存中就不会堆积起来,这是一种接近于数据流的小颗粒工作流。这样,就避免了因 spill 而写盘。当然,这里的前提就是不能有全局的排序。

由此可见,Spark 的所谓“内存计算”,是靠 DStream 才得以实现的,而 DSream 排除了全局排序的可能。这样一来,一方面避免了被动的因 spill 而来的写盘,另一方面还在“横向”的跨 Partition(Split 实质上就是 Partition)并行的基础上增加了“纵向”的跨时间先后的并行度。这两个因素显然有利于提高系统的计算速度并提高实时性。所以,如果把 Hadoop 和 Spark 放在一起比,用相同的数据和算法实测二者的计算速度,前者用 MapReduce,后者则用 DStream,那么毫无悬念后者会赢;但这是以不排序为前提的,Hadoop 的问题是没有给用户提供这样的选项。

其实 Hadoop 也有流模式的计算框架 Stream 和 Chain,在那种模式下,如果运用得当,也能得到很好的性能。但是一来要运用得当也并非易事,二来 Hadoop 的 Stream 和 Chain 相比之下确实不如 Spark 的 DStream 那样成熟和完整。Stream 和 Chain 之于 Hadoop,让人感觉好像是贴在 Hadoop 外面而非水乳交融成为一体。但是事情还有另一面,Hadoop 让用户可以利用 Unix/Linux 的那些 Utility 工具程序搭建分布式的数据流,这又是个优点。再说,针对某些特定目标的计算,Hadoop 的性能不输于 Spark 甚至反而胜出,那也是完全可能的。

20.7 使用的方便性

一个系统,要受到使用者的衷心喜爱而变得流行,其使用一定得要方便灵活。我们不妨回顾一下,当年的 Unix 之所以那么受欢迎,其中的原因之一就在于其 Shell 所提供的方便性和灵活性,具体就是其“管道(Pipe)”和输入输出重定向机制所带来的方便和灵活。如前所述,Unix 的管道机制(以及输入输出重定向机制)是搭建数据流的手段。这样的数据流搭建固然可以通过门槛相对较高也相对比较麻烦的 C 语言编程实现,同时又有方便得多、简单得多的方法加以实现。那就是通过 Shell 编程(作为脚本)实现,或者就直接通过命令行交互而实现,而且二者所使用的语言是一样的,都是集脚本语言与命令行语言于一身的 Shell 语言,如 bsh 或 csh。进一步,即使通过 Shell 编程实现,那也是解释执行的脚本而无须经过编译。对于使用者而言,特别是对于(编程语言)专业化程度不高的用户而言,解释执行和编译执行大不一样。当年的学生学 Fortran 和 Basic 两种语言,前者是编译的,后者是解释的,对于内行的人似乎难度上差别不大,但是实际上许多人都喜欢用 Basic 而视 Fortran 为畏途,至于 C 就更不用说了。所以,这跟所用的语言有很大的关系。适合于使用者的语言,无论是编程语言或命令行语言,其入门的“门槛”必须要低;至于复杂起来可以有多复杂,那倒问题不大,因为太复杂的就用 C 写了。事实上,作者至今还不能说是“熟练掌握 csh”,但是这并不妨碍我在键盘上输入几行 for 循环试一下,或者写个很简单的脚本,而且一般也已经够了。再要复杂一点的就查一下 man,在键盘上试几下,也就可以了。Unix 的那些 Shell,如 bsh、csh、bash、ksh、ssh 等,就都具有这种门槛低但“上不封顶”的特点。事实上,Unix 之所以广受欢迎,有这样集命令行语言和脚本语言于一身的 Shell 语言是个很大的因素。这样的机制,现在称为 REPL,

Read-Evaluation-Print Loop,即“读入—计算—打印”的循环,或者说交互式的解释执行。

光有一个门槛不高的 Shell 语言还是不够,Unix 还提供了许多工具性的小应用程序,称为 Utility,例如 cat、grep、wc、uniq、find、cpio 等。每个这样的小程序都既可以作为独立的应用运行,也可以嵌入数据流中作为一个节点运行,对操作系统而言都是独立的进程。这样,哪怕用户什么程序也不写,光在命令行或脚本中选择现成的小工具加以组合,就可以做很多事情了。但是当然,用户可以根据自己的需要开发更多这样的小程序,增添到系统的“武库”中。所以 Unix 系统中有 /bin、/sbin、/usr/bin 等目录,用来存放这些“兵器”。你每往这些武库中增添一件“兵器”,实际可以得到的是它与其他 Utility 程序的种种组合,即种种数据流。这样的程序设计风格,当时的人们就称之为“Unix 风格”。

反观 Hadoop,就不一样了。前面我们看过几个示例的代码,那都是用 Java 语言编程的。固然 Java 并不是很难用的语言,但是实际进行数据挖掘的往往不是程序员而是“数据科学家(Data Scientist)”,他们的特长不在编程,往往听见 Java 编程就头大,因为 Java 编程的门槛还是不低的,并非稍加训练就可解决。于是就得要给他们配上程序员作为助手,这样事情就复杂起来了。所以,Hadoop 缺少了一个友善的用户界面,没有提供一种友善的 REPL 机制,用 Java 语言搭建计算框架的门槛太高。

进一步,如果我们将 Hadoop 与 Unix 进行类比,那么与 Utility 程序相对应的应是各种各样现成的 Mapper 和 Reducer,或者各种各样现成的 map() 和 reduce() 函数,供用户选用。然而有没有呢?不能说没有,但是确实不多。这样,用户要在 Hadoop 上解决自己的问题,不仅要用 Java 语言编写用于搭建数据流的主控程序,还动不动就得开发自己的 Mapper 和 Reducer。这又进一步抬高了入门的门槛,由此而带来的麻烦很可能使用户望洋兴叹。

相比之下,Spark 就注意到了这些问题。Spark 出自加州大学 Berkeley 分校,那是当年 Unix 的“重镇”。不管是否来自 Unix 的直接影响,总之 Spark 采用了 Scala 作为其“Shell 语言”(也是 Spark 的编程语言),而 Scala 就是一种门槛不高的语言。Scala 既可用作粗粒度的 Shell 语言,用来构筑数据流,又可用作一般的、细粒度的编程语言,事实上 Spark 本身的代码主体就是用 Scala 语言编写的。进一步,Spark 还提供了许多现成的类和库函数,供用户选用。例如机器学习,Spark 就提供了不少现成的类,有个函数库 MLlib。当然,用户还可以结合自己的需求再进一步开发类似的类库和函数库。而 Scala 语言,则又支持将用户提供的类或函数扩展成其语言成分,这又与 Unix 的 Shell 相似。(试想,在命令行“cat file2 | sort | uniq | wc-l”中,sort 既非变量,也非预先定义的函数,形式上就像 Shell 的语言成分一样。Shell 对此的处理方法是:如果不是 Shell 语言中原始的关键字,也非变量或函数名,就去环境变量所指定的诸多目录中寻找是否有这样的可执行程序,如果有就创建一个进程加以执行。这样,例如 sort 就仿佛变成了 Shell 语言中的一个关键字,变成了 Shell 的语言成分之一。所以我们说 Shell 是一种开放的、“上不封顶”的语言。)

所以我们可以说,Spark+Scala 的组合继承了 Unix+Shell 的精髓。下面我们通过一个实例来说明为什么说 Spark 的用户界面比 Hadoop 友善,这个例子取自 *Learning Spark* 一书:

```
scala> val lines = sc.textFile("README.md") // Create an RDD called lines
lines: spark.RDD[String] = MappedRDD[...]    //这是 scala 打印出来的回应或结果,下同
scala> lines.count() // Count the number of items in this RDD
```

```
res0: Long = 127
scala> lines.first()// First item in this RDD, i.e. first line of README.md
res1: String = # Apache Spark
// To exit the shell, you can press Control + D.
```

这就是一个 REPL 的过程,你输入一行简单的命令(语句),马上就可以看到一些结果,错了就重来。这里的黑体字符都是用户的键盘输入,斜体字符都是 Spark 的输出,注释是另加的。

第一个命令行表示将文件 README.md 的内容读入缓冲区 lines,这就是一个 RDD。至于这个 RDD 分成几个 Partition,相当于 Hadoop 的输入分几个 Split,那要看文件的大小。

第二个命令行对这个 RDD 的内容实施 count() 计算,数一数文件中一共有多少行。这两个命令行也可以合并成一行:“sc.textFile("README.md").count()”。会用 Unix 的读者应该马上就能联想到“cat README.md | wc -l”,这二者的实质是一样的,只是形式不同。当然,在 Unix 的管道中,cat 和 wc 都是独立的进程,而 Spark 的 textFile() 和 count() 看似只是一般的函数调用;但是如果想到在一个数据流中 textFile() 和 count() 可以在不同的机器节点上执行,代表着数据流中的不同节点,那就没有多大区别了。另外,这里的 count() 与 reduce() 一样是个 Action,它的计算不是把一个 RDD 转化成另一个 RDD,而是从 RDD 算出一个(或一组)值,在这里就是 127。

Spark 对命令行的解析和执行是这样进行的:从构建第一个 RDD 开始,所构建的 RDD 就是一个 RDD 依赖序列的开头;然后,所给定的函数必须是这个 RDD 中有定义的函数,如果所给定函数的返回类型也是一种 RDD,那么所进行的计算是 RDD 的转化,即 Transformation,此时 Spark 只是用其构筑 RDD 依赖,并根据具体 RDD 的类型定义获知转化的结果是个什么类型的 RDD,但并不立即进行计算。如此类推,直至所给定函数的返回类型不再是 RDD,这就是个 Action,例如这里的 count(),这意味着 RDD 依赖序列的结束,此时就要提交作业并调度其运行了。这就是 Spark 的所谓“lazy evaluation”模式。其实也说不上什么“lazy”,这就好比命令行的构筑,在还没有按下回车键之前是在命令行的构筑阶段,那还是一个未完成的命令行;按下回车键,命令行的输入就完成了,下面就是执行的事了。RDD 依赖序列的构筑也是这样,这就好像命令行的构筑,Action 性质的函数则就像回车键。

所以,输入上面的第二个命令行后,计算就开始了,并且很快就得出结果为 127。

虽然这两个命令行可以合并成一个,但是像上面这样分成两行也有好处,而且是个很大的好处,就是可以把 sc.textFile() 的结果(那是一个 RDD)作为整个计算过程中的一个中间结果显式地赋给一个有命名、可引用的变量(Scala 语言中有两种变量, val 表示一经赋值便不可更改, var 表示赋值后仍可更改),那就是这里的 lines。

这样,正因为有了这个 lines,上面的第三个命令行中就可直接加以引用了。这个命令行对 lines 执行 first(),即摘取其第一行,就是“# Apache Spark”。注意,这个 first() 也是个 Action,所以也是一经输入就会立即被执行的,而对 lines 这个 RDD 的引用则另起了一个 RDD 依赖序列,lines 就是里面的第一个 RDD。这实际上就是前面那个 RDD 依赖序列上的一个分支,尽管也是马上就遇上了 Action。如果没有对 lines 这个 RDD 的引用,这里就还得再计算一次“sc.textFile("README.md")”。对比于 Unix 的管道机制,在那里你确实还得再做例如“cat README.md | first”。这意味着,相对于 Unix 管道机制那种链状的数据流,现在可以

在一个节点的输出端分叉,变成可以支持树状的 DAG(有向无环图)数据流了。

应该说,这是 Spark 对 Unix 管道机制的一项重要改进。至于 Hadoop,则早期的版本只支持 MapReduce,还谈不上分叉的问题;较新的版本则也支持“流(Stream)”模式,但那也只是利用 Unix 的管道机制而已,还是不能支持数据流分叉,自然不能说是支持 DAG。

这意味着,Spark 提供了结构上的更大灵活性,这本身就具有很大的意义;而从用户界面友善性的角度看,这也有利于降低使用门槛。在使用、学习的时候,用户往往是先在单机上调试,把程序调通了再提交到集群上去,这一点对于 Hadoop 和 Spark 都一样。但是在单机上运行 Hadoop 而要达到这个例子所示的效果,你至少得要写一些 Java 程序并加以调试;而如果是在单机上运行 Spark,那就简单多了,一步步键入命令行就可调试,调通了之后再汇集写在 Scala 脚本或程序中,就可以把它提交到集群上,这就降低了“门槛”。究其原因:一是 Scala 语言的作用;二是 Spark 提供了更多如 count()、first()之类现成的函数以供调用。可以想象,如果在 Unix 上没有 Shell 语言,而动不动都得写 C 程序,并且除 libc 外就没有多少库函数可供调用,那该有多么不便。

除 Scala 以外,Spark 也允许用 Python 作为其 Shell 语言,总归也比使用 Java 语言方便。

所以,比之 Hadoop,人们常常更喜欢用 Spark,这绝非偶然,是有原因的。

20.8 几个重要的类及其作用

明白了上述的这些原理,我们可以摘要看一下 Spark 中几个重要成分的类型定义,这将有利于我们加深对上述原理的理解。

首先当然是 RDD。在 Spark 的代码中,RDD 是个抽象类:

```
abstract class RDD[T: ClassTag]{}
] _sc: SparkContext    //_sc 的类型是 SparkContext
] id: Int              //每个 RDD 都有个整数型的 id
] dependencies_ : Seq[Dependency[_]]    //获取本 RDD 的 Dependency 序列
] partitions_ : Array[Partition]        //获取本 RDD 的 Partition 数组
] getPreferredLocations(split: Partition) //参数 split 是个 Partition,获取其偏好地点
] persist(newLevel: StorageLevel, allowOverride: Boolean) //持久化存储本 RDD
] compute(split: Partition, context: TaskContext) : Iterator[T] //通过计算获得本 RDD 的数据
] computeOrReadCheckpoint(split: Partition, context: TaskContext) //获得本 RDD 的数据
  > if (isCheckpointedAndMaterialized) {
  >+ firstParent[T].iterator(split, context) //如果有 Checkpoint 存在就直接使用
  > } else {
  >+ compute(split, context)                //没有 Checkpoint 就 compute()
  > }
] map[U: ClassTag](f: T => U) : RDD[U] //本 RDD 的 map()函数,返回类型为 RDD
  > cleanF = sc.clean(f)
  > new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.map(cleanF))
] filter(f: T => Boolean) : RDD[T]
//本 RDD 的过滤函数,符号“=>”表示从 T 到 Boolean 的类型转换
```



```

] coalesce(numPartitions: Int, shuffle: Boolean = false) : RDD[T]
  > if (shuffle) { //需要 shuffle
  >+ distributePartition = ...
  >+ // include a shuffle step so that our upstream tasks are still distributed
  >+ new CoalescedRDD(
    new ShuffledRDD[Int, T, T](mapPartitionsWithIndex(distributePartition),
    new HashPartitioner(numPartitions)), numPartitions).values
  > } else { //不需要 shuffle
  >+ new CoalescedRDD(this, numPartitions)
  > }

] sortBy[K](f: (T) => K, ...) : RDD[T] //这回类型为 RDD,所以这是一种转换
] reduce(f: (T, T) => T): T //返回类型不是 RDD,所以这是 Action
  > cleanF = sc.clean(f)
  > mergeResult = (index: Int, taskResult: Option[T])
  > sc.runJob(this, reducePartition, mergeResult)
  > jobResult.getOrElse(throw new UnsupportedOperationException("empty collection"))

] take(num: Int): Array[T] //Take the first num elements of the RDD.这也是 Action
] collect[U: ClassTag](f: PartialFunction[T, U]): Array[U] //返回该 RDD 中的所有元素
] count(): Long
] intersection(other: RDD[T]) : RDD[T] //两个 RDD 的交集
] cartesian[U: ClassTag](other: RDD[U]) : RDD[(T, U)] //两个 RDD 的笛卡尔乘积
] subtract(other: RDD[T]) : RDD[T] //两个 RDD 的差集

```

Spark 的代码是用 Scala 语言写的,要搞清这个摘要中的内容需要了解一些 Scala 的语义,但是 Scala 与 Java 多少也有点像,所以就这么看看也能知道个大概。在 Scala 语言中,变量名、函数名在前而类型在后,所以例如以“`map[U: ClassTag](f: T => U) : RDD[U]`”为例,就表示 `map()` 是个函数,其返回类型是 `RDD[U]`,即类型为 `U` 的数据的 RDD。而调用参数 `f` 则是个 Lambda 函数,这个函数进行从类型 `T` 到类型 `U` 的映射;换言之,在 `map()` 之前是个类型为 `T` 的 RDD,`map()` 之后则转化成类型为 `U` 的 RDD。这里的 `T` 和 `U` 均为泛型。

这里 `computeOrReadCheckpoint()` 和 `compute()` 这两个函数需要加一点说明。我们知道,整个计算流程是一个 Stage 的序列,或者说 Stage 的 Dependency,对其中最后一个 Stage 的最后一个 RDD 所做的计算必须是个 Action。Spark 要见到要求对一个 RDD 进行 Action 计算时才会提交作业,然后开始进行计算,而且逻辑上的起点就是这最后一个 RDD。但是此时这个 RDD 显然还没有数据,还是一片空白,它的数据要由它前面的那个 RDD 计算产生,所以才说是“依赖”于前面的那个 RDD。有了数据以后,才能对这些数据进行诸如 `reduce()`、`collect()`、`count()` 以及 `map()`、`filter()` 等等 Action 或 Transform 计算。同理,这个 RDD 的数据是靠对前一个 RDD 的 Transform 计算产生的,而调用当前 RDD 的 `compute()` 函数,其实就是要求计算产生这个 RDD 的数据,实际上就是要求启动对于前一个 RDD 的 Transform 计算。但是前一个 RDD 的数据也是空白,又依赖于它的前一个 RDD,于是又得调用它的 `compute()` 函数,就这样一直可以推到最前面的那个 RDD,那个 RDD 有数据。所以,实际的计

算当然是从前到后的。不过,也有可能中间的某个 RDD 就因为 Checkpoint 存在而无须向前追溯了,所以实际调用的应该是 `computeOrReadCheckpoint()`,先检查一下这个 RDD 是否有 Checkpoint,没有才需要 `compute()`。注意抽象类 RDD 并未提供具体的 `compute()` 函数,这要由扩充了 RDD 的具体类提供的,因为不同的具体 RDD 类获取其数据的方式也有所不同。

还有,这从前到后逐个 RDD 的计算,都是要等当前 RDD 中的数据都到位了之后才进行的,仍是批处理式的工作流而不是数据流,这跟 MapReduce 没有多大区别,但是排序却不再是强加的,而是显式地提供了一个函数 `sortBy()`,让用户选择。

注意上面这只是个摘要,定义于抽象类 RDD 的函数远不止这么一些。这里所列的有些是属于 RDD 内务的一些操作;再就是一些用于 RDD 转化的函数,例如 `map()`、`filter()` 等,这些函数的返回类型都是 RDD;然后是若干作为 Action 的函数,如 `reduce()`、`take()`、`count()` 等,这些函数的返回类型就不是 RDD 了;最后是若干用于集合计算即用来整合汇聚两个 RDD 的函数,如 `intersection()`、`cartesian()`、`subtract()` 等。

可见,Spark 的 RDD 提供了许多现成的计算方法让应用程序员选用,在这一方面 Hadoop 简直望尘莫及。

具体可以实体化的,当然是对这抽象类 RDD 的扩充,那就多了,下面只是略举数例:

```
class BlockRDD extends RDD{}
class MapPartitionsRDD extends RDD{}
class CoalescedRDD extends RDD{}
class NewHadoopRDD extends RDD{}
class KafkaRDD extends RDD{}
class SubtractedRDD extends RDD{}
class CartesianRDD extends RDD[Pair[T, U]](sc, Nil){}
class JdbcRDD extends RDD{}
class SqlNewHadoopRDD extends RDD{}
class EmptyRDD extends RDD{}
class ShuffledRDD extends RDD{}
```

最后这个 `ShuffledRDD` 显然就是经过 Shuffle 以后的 RDD。

而最前面的 `BlockRDD`,则是一种特殊的 RDD,由 `DStream` 所转化、生成出来的 RDD 就是 `BlockRDD`。

这些类对 RDD 的扩充,有的是在此基础上增添了若干数据成分或函数,有的是覆盖了 RDD 的某些函数,不一而足。不管怎么扩充,总而言之这些都是 RDD。

显然,抽象类 RDD 所描述的正是分布的数据集合,即若干 Partition 的集合,以及定义于该集合上的操作。而为某个 RDD 具体类创建的对象,则代表着一个具体的、特定的分布数据集合,也代表着数据流(工作流)中的一步计算,所以单个 RDD 就可以成为其中的一个计算节点。

RDD 不是凭空而来,总有个来历,除第一个 RDD 来自文件或通信接口外,其余的 RDD 都是来自另一个什么 RDD 的转化结果,也就是“依赖”于那个 RDD,这就是 RDD 的 Dependency。Spark 的代码中定义了一个抽象类 `Dependency`,然后又在这个基础上扩充成几个具体的 `Dependency` 类,用来描述 RDD 之间的依赖关系:

```

abstract class Dependency[T] extends Serializable {}
abstract class NarrowDependency[T] extends Dependency[T] {}
    //依赖有宽狭之分,这是狭依赖,表示 Partition 中数据的来源范围小
class OneToOneDependency[T] extends NarrowDependency[T] {}
    //最狭的依赖是一对一,即后续 RDD 中一个 Partition 的数据
    //只来自前导 RDD 中的一个 Partition、一个节点,因而无需 Shuffle
class RangeDependency[T] extends NarrowDependency[T] {}
    //这也是狭依赖,但基本不用,只是用在 Union 两个 RDD 的时候
class ShuffleDependency extends Dependency {}
    //这是宽依赖,后续 Partition 的数据来自多个节点,所以需要 Shuffle

```

可想而知,如果两个 RDD 之间的依赖是 OneToOneDependency,那就可以把它们安排在同一个计算节点上。这样的一串 RDD 就可构成一个 Stage。Spark 的代码中也定义了一个抽象类 Stage,然后扩充成具体类:

```

abstract class Stage extends Logging {}    //不用关心 Logging
] id: Int    //这个 Stage 的 id
] rdd: RDD[_]    //这个 Stage 中的 RDD 序列
] numTasks: Int    //这个 Stage 所含的任务数量
] parents: List[Stage]    //这个 Stage 前面的那些 Stage,是个 Stage 依赖序列
] firstJobId: Int    //这个 Stage 所属的第一个作业的 id
] callSite: CallSite    //要求创建这个 Stage 的 App 所在节点
] makeNewStageAttempt(numPartitionsToCompute: Int, ...)
    //试图调度执行这个 Stage 的计算,类似于 Hadoop 中的 TaskAttempt 之类

```

和 RDD 有个依赖序列一样,Stage 也有个依赖序列,就是 List[Stage]。Spark 的灵活性和功能上的优越性就在于可以构建出各种不同的 Stage 序列,而不只是 MapReduce。

比之 Hadoop 的 ApplicationAttempt 和 TaskAttempt,Spark 中多了个 StageAttempt,原因很简单。相比之下,Hadoop 中只有固定的两个 Stage,即 Map 和 Reduce,每个 Stage 中只有一个 Task,所以就不需要有这个概念了。

对抽象类 Stage 的扩充只有两种:

```

class ShuffleMapStage extends Stage {}
class ResultStage extends Stage {}

```

就是说,一共只有两种 Stage。一种是 ShuffleMapStage,就是对输入 RDD 进行种种转化,需要 Shuffle 的 Stage;另一种是 ResultStage,就是通过某种 Action 如 reduce()、count()等从输入 RDD 中产生(非 RDD)计算结果的 Stage。

可想而知,Stage 应该是像 Job、Task 一样受指派调度运行的单元,并且位于 Job 和 Task 之间。就是说,一个 Job 分解成若干个 Stage。在批处理的模式中这些 Stage 并不同时运行,因为前面的 Stage 不完成后面就没有数据,所以后面的 Stage 可以在前面的 Stage 完成之后才调度运行。事实上 Spark 的调度器就是这样调度的,DAGScheduler.submitJob()会被落实到 submitStage(),进一步又落实到一连串 submitMissingTasks();而且当有 Task 完成计算的时

候,如果这意味着整个 Stage 也因此而完成计算的话,就会触发下一个 Stage 的提交,事件处理函数 `handleTaskCompletion(event: CompletionEvent)` 中就可能调用 `submitStage()`。

而 Stage,则又分解成许多任务,其中有些任务可以放在同一节点上运行,但是绝大部分都得分布到 N 个节点上, N 就是具体 RDD 的分区个数。其实 Hadoop 也是这样做的,只是没有把例如 Sort 作为独立的 Task 向 App 程序员开放,并且一共只有 Map 和 Reduce 这么两个 Stage。

至此,我们已经大概明白了 Spark 对基于 RDD 的批处理流程,下面再看基于 DStream 的流处理。

在 Spark 的代码中,DStream 也是个抽象类,这是其类型定义的摘要:

```
abstract class DStream[T; ClassTag] (ssc: StreamingContext) extends Serializable {}
] slideDuration: Duration      //从输入流中切分数据生成 RDD 和作业的周期
] dependencies: List[DStream[_]] //DStream 的依赖序列
] storageLevel: StorageLevel    //中间结果的存储方式
] checkpointDuration: Duration //产生 checkpoint 的周期
] graph: DStreamGraph          //整个数据流/工作流的拓扑
] generatedRDDs = new HashMap[Time, RDD[T]] () //一个便查表,根据时间点查找 RDD
] compute(validTime: Time)      //类似于 RDD.compute()
] getOrCompute(time: Time)      //类似于 RDD.computeOrReadCheckpoint()
] checkpoint(interval: Duration) //创建 checkpoint
] generateJob(time: Time)       //从输入流中切分数据,生成一个作业
] map[U; ClassTag](mapFunc: T => U) //类似于 RDD.map(),下同
] flatMap[U; ClassTag](flatMapFunc: T => Traversable[U])
] filter(filterFunc: T => Boolean)
] repartition(numPartitions: Int)
] reduce(reduceFunc: (T, T) => T)
] count()
] transform[U; ClassTag](transformFunc: RDD[T] => RDD[U])
] print(num: Int)
] window(windowDuration: Duration, slideDuration: Duration) //构成 DStream 处理窗口
```

摘要中加了注释,结合前面的有关介绍,读者应该不难明白 DStream 中一些成分的作用。这里需要说一下 `generateJob()`。前面讲了,Spark 对于 DStream 的处理并非真正的流处理,而是把输入流按给定的时间周期切片后进行较小批量的批处理。事实上,切片后生成的 RDD 是 BlockRDD。但是这只解决了从 DStream 中切片生成 BlockRDD 这一环节,而没有解决对这个 BlockRDD 进行什么处理的问题。不言而喻,对每一个具体切片 BlockRDD 的处理都应该与用户所要求的对 DStream 的处理一致。如果把用户所要求的对 DStream 的处理流程看成一个宏观意义上的、笼统的“作业”;那么对每个切片所做的、相同拓扑的处理流程就是一个微观的、具体的作业。而这里的函数 `generateJob()`,就是供调用用来生成这样的作业。所以,对于一个 DStream,Spark 每过一会儿就要生成一个作业,包括其 RDD 序列,并通过 `submitJob()`

提交运行。当然,这又会分解转化成 `submitStage()`,然后又有 `submitMissingTasks()`,最后投运的是 Task,这又跟 MapReduce 框架中一样。Spark 的 Task 有两种,一种是 `ShuffleMapTask`,另一种是 `ResultTask`。虽然,Hadoop 中的 MapTask 其实是一种 `ShuffleMapTask`,而 `ReduceTask` 其实是一种 `ResultTask`。可见,Spark 在 Hadoop 的基础上做了推广。

值得指出的是,对于 DStream,每当生成出一个新的作业时,此前的作业未必已经运行完成,最大的可能是还在计算、处理其中的某一个 RDD。而新的作业,只要集群中还有资源,就无须等待前面的作业完成就可开始运行,于是就形成了由多个 DStream 切片构成的“窗口”。这个窗口的内容随着时间而推进,成为所谓的“滑动窗口”。这样,与一般的 RDD 相比,对 DStream 的计算就增加了时间先后维度上“纵向”的并行性,变得更像数据流,更像流水作业了。一般 RDD 的那种“横向”的并行存在于同一 RDD 的不同 Partition(Split 就是 Partition)之间,而 DStream 所增加的“纵向”并行存在于这样的不同作业之间,或者同一滑动窗口内的不同 RDD 之间。DStream 使 Spark 对于数据的处理方式更接近于“流处理”而不是“批处理”。

不过,流处理与批处理之间,或者说数据流与工作流之间,也并无绝对的高低优劣之分。何者更为合适要看具体应用的性质、特点和要求。如前所述,那只是通信传输的粒度不同,当粒度小到一定程度时,通信的开销就会上升到不可承受或至少是不划算的地步。

至于性能上的比较,有的说 Spark 能比 Hadoop 高出 100 倍,少说也能高出 10 倍;反过来也有人说进行某些计算时 Hadoop 反倒更快。双方也都有实测的数据为证。其实他们都没有错,不同框架与平台的性能好坏与具体进行什么样的计算和处理以及对结果的要求有着密切的关系,他们那些实测的数据都是有条件的,双方都能找出一些特别适合自身特点的计算来显示己方的性能优越。

但是一般而言,根据上面的介绍和分析,Spark 的性能应该比 Hadoop 好一些。如果是对一般 RDD 的处理,因为同样是批处理,并行度相仿,与 MapReduce 本应相差不大;但是在 Spark 中是否排序是可选的,如果跳过排序这个环节,那么性能当然会有提高。而如果采用 DStream,那就有了“纵向”的并行,整个计算的并行度可以得到显著提高,对于性能的改进就不用说了。

不过二者的比较并非只是性能上的,事实上这些年来 Hadoop 已经形成了它的“生态系统”,在 Hadoop 的基础上已经派生出许多别的软件,如 HBase、Hive、Tez、Pig、Mahout 等,其实在某种意义上连 Spark 本身也属于 Hadoop 的生态系统。从性质上看,Hadoop 更接近于一个(集群的)操作系统,而 Spark 则是单纯的应用,但是一般而言效率比较高一些。所以,客观上现在 Hadoop 和 Spark 都已成为大数据处理系统的事实标准。

这么说下来,读者也许会问:既然如此,那你这本书为什么不干脆就写 Spark,而要写 Hadoop 呢?其实我也这样考虑过,但是觉得跳过 Hadoop 而直接写 Spark 其实是不太现实的,那样势必时时都要补叙 Hadoop 的内容,最后就成为二者的叠加,这就好像房子不能没有一楼就直接造二楼一样。但是读者已经看到,光是 Hadoop 就有这么厚厚一本,再要加上对 Spark 代码的深度分析,就不合适了。反过来,如果读者把 Hadoop 搞懂了,在此基础上要自己进一步研究 Spark,那就不会很难。再说,类似的大数据处理系统也不只是 Spark 一种,例如 Storm 就是更适合用于实时处理的系统。所以,更重要的是掌握大数据处理系统的一些基本的思路和方法。若是以此为目的,则以 Hadoop 的代码为标本加以解剖和分析,就再合适不过了。

参考资料

【参考源码和网站】

[1] Hadoop 2.6.0 源码:hadoop-2.6.0-src.tar.gz,发布于 2015/2/5

注:目前最新的版本是 Hadoop 3.0 alpha 2,中间主要经历 Hadoop 2.7 (2016/7/20)

[2] Spark 1.6.2 源码:spark-1.6.1.tgz,发布于 2016/1/4

注:目前最新的版本是 Spark 2.1.0,中间主要经历 Spark 2.0 (2016/7/26)

[3] Hadoop网站:<http://hadoop.apache.org/>

[4] Spark网站:<http://spark.apache.org/>

[5] ZooKeeper网站:<http://zookeeper.apache.org/>,源码:zookeeper-3.4.5.tar.gz

[6] OpenJdk网站:<http://openjdk.java.net>,源码:jdk-7u25-linux-x64.tar.gz

[7] Guice 网站:<https://github.com/google/guice>,源码:guice-3.0.zip

[8] JAAS网站:<http://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASRefGuide.html>

【主要参考文献】

[1] WHITE T. Hadoop: The Definitive Guide [M]. 2th ed, 3th ed, 4th ed. California: O'Reilly, 2010, 2014, 2015

[2] LAM C. Hadoop in Action [M]. New York: Manning, 2010

[3] MURTHY A, VAVILAPALLI V, EADLINE D, et al. Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2 [M]. New Jersey: Addison-Wesley, 2014

[4] KARAU H. Learning Spark [M]. California: O'Reilly, 2015

[5] LINDHOLMT. The Java Virtual Machine Specification, Java SE 7 Edition [R], 2013

[6] WARBURTON R. Java 8 Lambdas [M]. California: O'Reilly, 2014

[7] HWANGK. Computer Architecture and Parallel Processing [M]. New York: McGraw-Hill, 1985

责任编辑：樊晓燕 吴昌雷

封面设计： 春天·书装工作室

大数据处理系统

Hadoop 源代码情景分析

这本书并不是为所有想要对大数据有所了解的人而写的。但是，如果你有点野心，想对大数据处理系统有比较深入、透彻的了解，特别是想有朝一日自己也设计一个这样的系统，甚至自己把它写出来，那么你真应该认真读一下这本书，看看人家Hadoop是怎么设计怎么实现的。然后，在最后一章，你可以再看看Spark又是怎样的，有些什么改进。你将看到，在一个计算机集群上构筑一个大数据处理系统，哪些成分是必不可少的，哪些方面又是可以改进的，它与操作系统的关系怎样，而作为大规模计算机集群的“操作系统”又可以并应该是什么样的。

ISBN 978-7-308-16669-0



定价：128.00元